

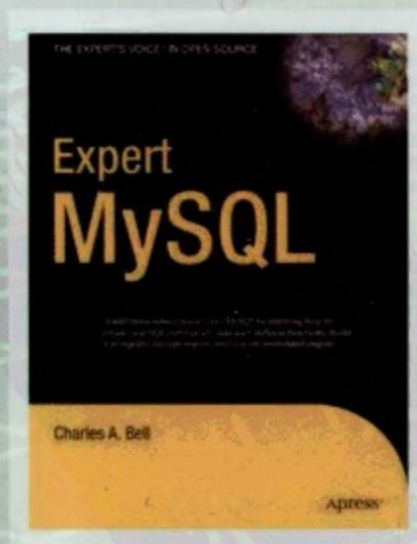
Expert MySQL

深入理解MySQL

[美] Charles A. Bell 著

杨涛 王建桥
杨晓云 韩兰 等译

- MySQL核心开发人员力作
- 带你深入MySQL源代码和底层架构
- 身临其境，透彻掌握数据库理论与实践



The Turing logo, featuring the word "TURING" in a bold, sans-serif font inside a black rectangular box.

图灵程序设计丛书 数据库系列

Expert MySQL

深入理解MySQL

[美] Charles A. Bell 著
杨涛 王建桥 杨晓云 韩兰 等译

人民邮电出版社
北京

www.TopSage.com

图书在版编目 (CIP) 数据

深入理解 MySQL / (美) 贝尔 (Bell, C. A.) 著; 杨涛
等译. —北京: 人民邮电出版社, 2010.1
(图灵程序设计丛书)
书名原文: Expert MySQL
ISBN 978-7-115-18910-3

I. 深… II. ①贝…②杨… III. 关系数据库-数据库管
理系统, MySQL IV. TP311.138

中国版本图书馆CIP数据核字 (2008) 第150271号

内 容 简 介

本书深入源代码, 剖析了 MySQL 数据库系统的架构, 并提供了分析、集成和修改 MySQL 源代码的专家级建议。本书分三个部分: 第一部分介绍开发和修改开源系统的概念, 提供探讨更高级数据库概念所需的工具和资源; 第二部分讨论 MySQL 系统, 阐明如何修改 MySQL 源码, 如何将 MySQL 系统作为嵌入式数据库系统; 第三部分更深入地探讨了 MySQL 系统, 讲述数据库工作的内部机理。

本书面向 MySQL 数据库开发人员。

图灵程序设计丛书

深入理解MySQL

-
- ◆ 著 [美] Charles A. Bell
 - 译 杨 涛 王建桥 杨晓云 韩 兰 等
 - 责任编辑 傅志红
 - 执行编辑 傅尔也
 - ◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
邮编 100061 电子函件 315@ptpress.com.cn
网址 <http://www.ptpress.com.cn>
北京顺义振华印刷厂印刷
 - ◆ 开本: 800×1000 1/16
印张: 30
字数: 803千字 2010年1月第1版
印数: 1-3 000册 2010年1月北京第1次印刷
著作权合同登记号 图字: 01-2007-4260号
ISBN 978-7-115-18910-3/TP
-

定价: 65.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154

前言

MySQL已被公认为是世界上最流行的开源数据库产品和行业内增长最快的数据库系统之一。来自MySQL AB公司^①的统计报告显示, MySQL的安装数量已超过800万, 每天的下载量接近5万人次。MySQL正迅速成为系统集成商首选的数据库系统。据SD Times报上的一篇文章报道, 对900多位读者进行的调查表明, MySQL在“装机量最大的数据库”榜上排名第三 (www.mysql.com/why-mysql/marketshare/)。

本书对数据库系统的一些高级问题进行了探讨, 对MySQL的体系结构进行了剖析, 还为分析、集成和修改MySQL源代码使之用于企业级环境提供了专家级建议。在如何修改MySQL系统来满足系统集成商和教育科研机构的独特需求方面, 本书提出了独到的见解。

本书结构

本书分为三个部分, 每个部分对一组彼此相关的问题进行探讨, 内容从MySQL和开源运动的发展, 到扩展和定制MySQL系统, 甚至还讲述了如何建立一个实验性查询优化器和执行引擎来替代MySQL查询引擎等。

第一部分

本书的第一部分对开发和修改各种开源系统所涉及的基本概念进行了介绍。这一部分为探讨本书后面介绍的更高级的数据库概念提供了必需的工具和资源。

与本书的其他章相比, 第1章对技术性问题的探讨相对较少, 多是些叙述性的内容。这一章的目的主要是让大家了解开源系统集成商都有哪些权益和责任。这一章突出介绍了MySQL的快速成长及其在开源和数据库系统市场中的重要性。此外, 这一章还清晰地勾勒出了开源运动发展的脉络。

第2章对什么是数据库系统和怎样构造数据库系统等基础知识进行了介绍。对MySQL系统的剖析充分展示了现代关系数据库系统的关键组件。

第3章对MySQL软件的源代码以及如何获得和构建一个MySQL系统做了全面的介绍。主要内容包括MySQL源代码的内部机制以及编码指导原则和如何维护源代码的最佳实践。

第4章介绍了生成高质量MySQL系统扩展的一个关键方面。这一章讲解了软件测试技术以及测试大型软件系统常用的实践方法, 采用几个具体示例展示了几种已被广泛接受的测试MySQL系统的方法。

^① 2008年1月, MySQL AB公司被Sun公司收购。2009年4月, Sun公司被Oracle公司收购。——编者注

第二部分

第二部分采用实际操作的方法来研究MySQL系统。这一部分介绍如何修改MySQL代码，以及如何把MySQL系统用作嵌入式数据库系统。还通过各种示例和项目向读者演示如何调试源代码，如何修改SQL命令来扩展这种语言，以及如何创建定制的存储引擎。

第5章介绍了一些调试技巧和技术，有助于保证开发工作更容易，减少不必要的错误和麻烦。在介绍各种调试技术的时候，还对它们的优缺点进行了分析和说明。

第6章指导读者掌握如何把MySQL系统嵌入企业级应用程序。这一章的示例项目将帮助读者运用学到的技巧来进行系统集成。

第7章是本书探讨MySQL代码修改问题的第一章。这一章演示了几种只需修改少量的MySQL代码就可以达到目的的技术。重点探讨MySQL的插件式存储引擎的能力，并通过有关的示例和项目构建一个示范性的存储引擎。

第8章介绍了最流行的MySQL代码修改技术。向读者展示了如何修改SQL命令以及如何建立定制的SQL命令。这一章给出了几个例子说明如何修改SQL命令以添加新参数、新函数和新命令。

第三部分

第三部分深入MySQL系统的内部去探查这个系统的工作原理。首先介绍了一些高级的数据库技术，精辟阐述了有关理论和实践，使读者能够运用所学到的知识去解决与数据库系统有关的更为复杂的问题。这一部分还给出了一些例子，介绍如何实现内部查询表示，如何实现新的查询优化器，以及如何实现新的查询执行机制。并对有关的示例和项目作了详细的讨论。第10~12章演示了如何改变MySQL系统的内部结构，以实现新的查询处理机制。这几章为如何建立和修改大型系统提供了独到的见解。

第9章介绍一些高级的数据库技术并对MySQL体系结构进行深入分析。主要内容包括查询执行、多用户问题以及编程时的注意事项等。

第10章讨论MySQL的内部查询表示，介绍了一个新的示例查询表示。主要讨论了如何通过修改MySQL源代码来实现新的查询表示。

第11章探讨了MySQL内部查询优化器，介绍一个示例性的新的查询优化器，这个查询优化器使用了第10章实现的新的查询表示。读者可以学会如何通过修改MySQL源代码来实现一种新的查询优化器。

第12章把前几章介绍的技术结合起来，指导读者修改MySQL系统来实现一种新的查询处理引擎技术。

附录

本书的附录列出了一份MySQL、数据库系统和开源软件的资源清单。

将本书作为讲授数据库系统内部结构的教材

介绍关系数据库理论和实践的优秀教材有很多。但是，适用于课堂教学和实验环境的资料并不多见，能帮助学生钻研数据库系统内部工作原理的资源就更少了。本书为那些通过实际动手实验来充实

其数据库课程内容的教师提供了一个机会。在课堂上使用本书的方式有三种。

首先，本书可以用来增加本科生或研究生的数据库初级课程的深度。本书的第一部分和第二部分对数据库系统的一些特殊主题进行了深入的讲解。推荐将第2、3、4章和第6章的内容作为授课主题，这几章的主题可当作对更为传统的数据库理论或数据库系统课程的补充。学生动手实践和课堂项目可以从第6章和第8章节选。

本书的第一部分和第二部分内容可用来开设一门本科生和研究生的高级数据库课程，这两个部分里的每一章都适用于课堂教学，可在8~12周讲完，多出来的授课时间可以用来讨论物理存储层的实现问题或加深对存储引擎的理解。学期项目可以以第7章为基础，让学生自行构建一个存储引擎。

面向高年级本科生或研究生的数据库系统高级专题课程可以使用本书作为基本教材，并把本书的前9章内容当作课堂教学的基础。学期项目可以借鉴本书第三部分内容，让学生为一个实验性数据库平台实现它还缺少的功能，包括语言理论、查询优化器、查询执行算法的应用。

开始行动吧

本书充分考虑了各类读者的需求。不论是与数据库系统已经打过多年交道，还是只听了一门数据库理论基础课，甚至只读过Apress公司出版的某本优秀的MySQL书，相信你都可以从本书学到许多东西。如果你想了解像MySQL这样的数据库系统是如何运转的，你甚至可以从源代码入手！

致谢

Apress出版公司到处都是充满天赋而又精明强干的专业人士，我要感谢他们当中的许多人。本书的编辑Jason Gilmore和项目经理Tracy Brown Collins都有着极大的耐心和非凡的见地。正是因为他们的努力，本书才能如期完成，让我言而有信。我还要感谢本书的生产编辑Katie Stence和文字编辑Liz Welch，他们让本书的印刷效果看起来相当不错。非常感谢两位！

我还要特别感谢以下几位技术审稿人：L. M. Parker和Mikael Ronström，正是他们毫不松懈地严格把关才保证了本书的质量；还有Michael Kruckenberg，他保证了本书编程示例的正确性，他对MySQL独特的见解和丰富的经验让我非常佩服。可以说，我曾与精英中的精英一起合作。

最后，我还要感谢我妻子Annette无尽的耐心和理解。

目 录

第一部分 MySQL 开发入门

第 1 章 MySQL 与开源运动	2	2.3.7 文件访问	33
1.1 什么是开源软件	2	2.3.8 查询结果	35
1.1.1 为什么要使用开源软件	4	2.3.9 关系数据库的体系结构小结	35
1.1.2 开源软件是否对商业软件构成 真正的威胁	7	2.4 MySQL 数据库系统	35
1.1.3 法律问题与《GNU 宣言》	8	2.4.1 MySQL 系统体系结构	36
1.1.4 将开源进行到底	10	2.4.2 SQL 接口	37
1.2 用 MySQL 进行开发	11	2.4.3 解析器	38
1.2.1 为什么修改 MySQL	13	2.4.4 查询优化器	39
1.2.2 MySQL 里哪些可以修改, 有什么 限制	14	2.4.5 查询的执行	40
1.2.3 MySQL 的许可证问题	15	2.4.6 查询缓存	40
1.2.4 到底能否修改 MySQL	16	2.4.7 缓存和缓冲区	42
1.2.5 修改 MySQL 的指导原则	17	2.4.8 通过插件式存储引擎访问文件	43
1.3 实际的例子: TiVo	18	2.5 小结	50
1.4 小结	19	第 3 章 MySQL 源代码	51
第 2 章 数据库系统剖析	20	3.1 预备知识	51
2.1 数据库系统的体系结构	20	3.1.1 了解许可证	51
2.2 数据库系统的类型	20	3.1.2 获得 MySQL 源代码	52
2.2.1 面向对象数据库系统	20	3.2 MySQL 源代码	56
2.2.2 对象关系数据库系统	21	3.2.1 预备知识	57
2.2.3 关系数据库系统	23	3.2.2 main() 函数	59
2.3 关系数据库系统的体系结构	24	3.2.3 处理连接和创建线程	62
2.3.1 客户端应用程序	25	3.2.4 解析查询	69
2.3.2 查询接口	26	3.2.5 优化查询的准备工作	75
2.3.3 查询处理	27	3.2.6 优化查询	78
2.3.4 查询优化器	29	3.2.7 执行查询	80
2.3.5 查询的内部表示	31	3.2.8 辅助库	82
2.3.6 查询的执行	32	3.2.9 重要的类和结构	83
		3.3 编程指导	88
		3.3.1 总体指导	89
		3.3.2 文档	89
		3.3.3 函数和参数	91

3.3.4 命名约定	92	6.1.2 嵌入式系统的种类	163
3.3.5 分隔与缩进	92	6.1.3 嵌入式数据库系统	163
3.3.6 文档工具	93	6.2 嵌入 MySQL	164
3.3.7 保持工作记录的习惯	95	6.2.1 嵌入 MySQL 的方法	165
3.3.8 追踪变化	95	6.2.2 资源要求	167
3.4 第一次构建系统	97	6.2.3 安全问题	167
3.5 小结	100	6.2.4 嵌入 MySQL 的优点	167
第4章 测试驱动的 MySQL 开发	101	6.2.5 嵌入 MySQL 的局限性	168
4.1 背景知识	101	6.3 MySQL C API	168
4.1.1 为什么要测试	101	6.3.1 预备知识	168
4.1.2 基准测试	103	6.3.2 最常用的函数	169
4.1.3 性能分析	105	6.3.3 创建嵌入式服务器	170
4.1.4 软件测试简介	107	6.3.4 对服务器进行初始化	171
4.1.5 功能测试与缺陷测试	107	6.3.5 设置选项	172
4.2 MySQL 测试	111	6.3.6 连接到服务器	172
4.2.1 MySQL Test Suite	111	6.3.7 运行查询命令	173
4.2.2 MySQL 基准测试	119	6.3.8 检索查询结果	174
4.2.3 MySQL 性能分析	124	6.3.9 清理	175
4.3 小结	126	6.3.10 与服务器断开连接并关闭 服务器	175
第二部分 扩展 MySQL		6.3.11 汇总	175
第5章 调试	128	6.3.12 出错处理	177
5.1 调试介绍	128	6.4 构建嵌入式 MySQL 应用程序	177
5.2 调试技术	129	6.4.1 编译 libmysqld 库	177
5.2.1 基本过程	129	6.4.2 调试问题如何解决	178
5.2.2 内嵌调试语句	131	6.4.3 数据问题如何解决	180
5.2.3 出错处理器	134	6.4.4 创建基本的嵌入式服务器	180
5.2.4 外部调试器	135	6.4.5 出错处理问题如何解决	189
5.3 调试 MySQL	142	6.4.6 嵌入式服务器应用程序	189
5.3.1 内嵌调试语句	143	6.5 小结	214
5.3.2 出错处理器	148	第7章 创建自己的存储引擎	215
5.3.3 在 Linux 环境里调试 MySQL	148	7.1 MySQL 插件式存储引擎概述	215
5.3.4 在 Windows 环境里调试 MySQL	157	7.1.1 基本过程	217
5.4 小结	161	7.1.2 需要用到的源文件	218
第6章 嵌入式 MySQL	162	7.1.3 其他辅助资源	218
6.1 构建嵌入式应用	162	7.1.4 handlerton 类	218
6.1.1 什么是嵌入式系统	162	7.1.5 handler 类	221
		7.1.6 对 MySQL 存储引擎的简要 分析	225

7.2	Spartan 存储引擎	226	9.3.3	实验项目的组成部分	342
7.2.1	底层 I/O 类	227	9.3.4	在 Linux 平台上进行实验	343
7.2.2	预备知识	250	9.3.5	在 Windows 平台上进行实验	343
7.2.3	阶段 1: 生成引擎存根	251	9.4	小结	343
7.2.4	阶段 2: 处理表	262	第 10 章	内部查询表示	344
7.2.5	阶段 3: 数据的读/写	269	10.1	查询树	344
7.2.6	阶段 4: 数据的更新和删除	273	10.1.1	查询转换	346
7.2.7	阶段 5: 数据的索引	280	10.1.2	DBXP 查询树	347
7.2.8	阶段 6: 添加事务支持	299	10.2	在 MySQL 里实现 DBXP 查询树	348
7.3	小结	303	10.2.1	被添加和修改的文件	349
第 8 章	为 MySQL 添加函数和命令	305	10.2.2	创建测试	349
8.1	添加用户定义函数	305	10.2.3	为 SELECT DBXP 命令生成 存根	350
8.1.1	CREATE FUNCTION 命令的 语法	305	10.2.4	添加查询树类	357
8.1.2	DROP FUNCTION 命令的 语法	306	10.2.5	显示查询树的细节	366
8.1.3	创建用户定义库	306	10.3	小结	373
8.1.4	添加新的用户定义函数	311	第 11 章	查询优化	374
8.2	添加本机函数	315	11.1	查询优化器的类型	374
8.2.1	在 Windows 平台上生成词法 散列表	318	11.1.1	基于开销的优化器	375
8.2.2	在 Linux 平台上生成词法散列 表	318	11.1.2	启发式优化器	377
8.2.3	编译和测试新的本机函数	318	11.1.3	语义优化器	377
8.3	添加 SQL 命令	319	11.1.4	参数优化器	378
8.4	添加到信息模式	327	11.2	再次讨论启发式优化	378
8.5	小结	333	11.3	DBXP 查询优化器	379
第三部分	高级数据库的内部组成		11.3.1	测试设计	379
第 9 章	数据库系统的内部组成	336	11.3.2	为 SELECT DBXP 命令生成 存根	380
9.1	查询执行	336	11.3.3	重要的 MySQL 结构和类	382
9.1.1	重温 MySQL 查询执行	336	11.3.4	DBXP 辅助类	385
9.1.2	什么是已编译查询	337	11.3.5	修改现有代码	386
9.2	深入 MySQL 的内部	337	11.3.6	启发式优化器的细节	390
9.2.1	开始用 MySQL 做实验	338	11.3.7	代码的编译和测试	413
9.2.2	注意事项	340	11.4	小结	417
9.3	数据库系统内部组成实验	340	第 12 章	查询执行	418
9.3.1	为什么叫实验	341	12.1	回顾查询执行	418
9.3.2	实验项目概述	341	12.1.1	投影	418
			12.1.2	限制	419
			12.1.3	联结	419

4 目 录

12.2 DBXP 查询执行	429	12.2.4 代码的编译和测试	454
12.2.1 测试的设计	430	12.3 小结	457
12.2.2 更新 SELECT DBXP 命令	431	附录	459
12.2.3 DBXP 算法	433		



Part 1

第一部分

MySQL 开发入门

这一部分介绍了一些与开源系统的开发和修改有关的基本概念。第1章介绍开源系统集成商有哪些权益和责任，重点讲述了MySQL的快速成长及其在开源和数据库系统市场中的重要性。第2章对什么是数据库系统以及如何构造数据库系统等基础知识进行介绍。第3章对本书所涉及的MySQL源代码以及如何获得和建立MySQL系统作了全面介绍。第4章介绍为MySQL系统生成高质量扩展的一个关键部分。读者在这一部分将学到软件测试技术以及一些测试大型软件系统的常用方法。

本 部 分 内 容

- 第 1 章 MySQL 与开源运动
- 第 2 章 数据库系统剖析
- 第 3 章 MySQL 源代码
- 第 4 章 测试驱动的 MySQL 开发

开源系统正迅速成为改变软件前景的一支重要力量。开源厂商在软件开发和技术支持方面展现出来的高品质，已成为全世界的信息技术专家所关注和议论的焦点，有许多开源厂商的产品和服务都达到了世界一流的水准。大公司在关注着，因为开源软件能够让它们第一次有机会摆脱商业软件厂商的专利产品；小企业也在关注，因为开源软件可以大幅降低它们在信息系统方面的成本开支；开发人员也在关注，因为开源软件让他们有了更大的挑选余地。在现今的因特网上，有许许多多的网站都是在Linux、Apache HTTP服务器、BIND、Sendmail、OpenSSL、MySQL和其他一些开源软件的基础上搭建起来的。

促使人们选用开源软件最常见的商业理由是成本。开源软件在从根本上减少软件产品的总体拥有成本（Total Cost of Ownership, TCO）的同时，还提供了一种可行的商业模式，让形形色色的企业去构建和完善它们的市场。具体到开源数据库系统，这一点体现得尤其明显。商业化专利数据库系统少说也要几千美元才能买到，如果加上技术支持方面的费用，总成本往往很容易就达到数万甚至数十万美元。

在过去，有许多人认为开源软件只限于爱好者或黑客们使用，而这些人目的不过是扰乱大型商业软件公司的市场而已。应该承认，有些开发人员觉得自己就是站在微软巨人面前的大卫^①，但开源社区与这些人完全是两回事。开源社区不以彻底取代某种商业专利软件为目标，他们对“开源”的解释是“向人们提供另外一种选择”。本章将向你展示，开源软件不仅让人们在商业软件之外多了一种选择，还在软件的开发和营销方面掀起了一场世界性的革命。

注解 在本书中，黑客这个词的含义同理查德·斯托曼（Richard Stallman）对黑客的定义：“热爱编程并为自己的聪明才智感到骄傲的人。”这与报章杂志上面提到的专门偷盗信用卡和损坏他人计算机系统的坏人有着本质的区别。

下面一节是为那些对开源软件或MySQL基本哲理还不甚明了的人准备的。如果你已经很熟悉开源软件的基本哲理，就可以直接跳到1.2节。

1.1 什么是开源软件

开源软件是人们有意识地抵制软件专利思维的产物。理查德·斯托曼在就职于MIT（Massa-

^① 圣经中，牧羊人大卫杀死了巨人哥利亚。这里借指某些开发人员认为他们是微软权威技术方面的克星。

chusetts Institute of Technology, 麻省理工学院) 的人工智能实验室时, 曾于20世纪70年代发起过一场代码共享运动。斯托曼当时的出发点是希望常用的代码能够为全世界的程序员所共享, 而这意味着开发人员可以在全世界的范围内彼此合作。这一理念在斯托曼和他的圈内朋友中贯彻得很好——当然, 这一切都发生在软件业集体决定软件属于专利并禁止与潜在的竞争对手进行共享之前。在那之后, MIT的许多研究人员因为各种原因离开MIT进入了这些软件公司。最终, 这个原本合作无间的小集体消亡了。

幸好, 斯托曼没有随波逐流, 他留在MIT并创立了GNU (GNU's Not Unix) 项目和自由软件基金会 (Free Software Foundation, FSF)。GNU项目的目标是推出一个没有专利束缚的Unix风格的操作系统。这个系统是自由的, 并对所有人开放 (包括对源代码的访问)。在这里, “自由” 的概念是不禁止任何人使用和修改这个系统。

斯托曼的目标是重新组建一支能够像他们当初在MIT那样合作的开发人员社区。可是, 斯托曼预见到这个系统需要一种版权许可证机制来保证一定程度上的自由。(据说, 斯托曼用 “copyleft” 而不是 “copyright” 来表达自己的对这种版权意义的理解, 它的含义是保证软件能够自由交流而不是受到限制。) 斯托曼起草了GPL (GNU Public License, GNU公共许可证)。GPL是一份闪耀着智慧光芒的法律许可文书, 它允许代码能够不受限制地进行复制和修改, 但要求衍生出来的软件产品 (以及被修改过的副本) 在发行时必须遵守与原始版本完全相同的许可证, 不得增加任何限制性条款。本质上来讲, GPL把专利因素全部排除在外, 收到了利用版权法来抵制版权的效果。

遗憾的是, 斯托曼的GNU项目最终还是没有完全成功, 但它的几个阶段性成果已经成为许多开源系统的基本元素, 其中最为成功的包括GNU的C语言编译器 (GCC) 和GNU文本编辑器 (Emacs)。虽然GNU操作系统没能完成, 但与斯托曼志同道合的人越来越多。李纳斯·托沃兹 (Linus Torvalds) 在斯托曼及其追随者的前期工作基础上终于填补了这一空白, Linux操作系统在1991年诞生了。Linux已发展成为一种自由的Unix风格的操作系统, 斯托曼的心愿总算实现了。如今, Linux已是世界上最流行、最成功的开源操作系统了。

Linux为什么如此流行

Linux是一种建立在开源模型上的Unix风格的操作系统。任何人都可以免费使用、传播和修改。Linux采用了一种非常保守的设计方案, 精简了其内核部分的功能, 因此很容易演化和完善。自1991年问世以来, Linux在全世界范围内得到了开发人员的广泛支持, 人们一直在改善它的性能和可靠性。有些人甚至宣称Linux是发展最成熟的操作系统。正式发布以来, Linux已在全世界的服务器和工作站市场上赢得了巨大的份额。如今, 人们提到Linux操作系统的时候都会把它当作最成功的开源代码成果来谈论。

自由软件运动在其发展过程中也曾遇到过问题。这里所说的 “自由” (free) 指的是人们可以自由地使用、修改和传播有关软件, 不是 “免费” 或 “没有成本” 的意思 (请大家体会 “自由发言” 和 “免费啤酒” 的区别)。为了解决这个问题, 那些先驱们组建了OSI (Open Source Initiative, 开源倡议), 并决定采用 “开源” 这个短语来描述GPL许可证所蕴涵和维护的 “自由” 含义。^①对此感兴趣的读者

① 需要注意的是斯托曼本人并不认可开源软件等同于自由软件。——编者注

可以到www.opensource.org网站上去看一看。

OSI组织的努力改变了自由软件运动。软件开发人员开始认识到自由软件并非不需要支付成本，而开放软件也只有懂得与他人合作的团体和个人才能合法使用。随着因特网的迅速普及，合作性社区已经成为了一个世界性的开发人员社区。这个世界性的开发人员社区确保斯托曼的初衷能够得以延续。

因此，开源软件是这样一种软件，它用许可证制度确保开发人员在参与某个合作性社区（该社区的根本目的是让高质量的软件能够繁荣发展）的开发时，有权自由地使用它并对其进行复制、修改和传播。“开源”并不意味着零成本，它实际上意味着任何人都能参与某个软件的开发，因而可以无须支付任何费用使用那个软件。事实上，许多开源系统都依托于某个组织，由该组织发布，而该组织会通过为相关软件提供收费的技术支持服务来维持运转。这种模式下，使用该软件的组织可以消除启动成本，大幅减少软件维护费用来降低信息技术方面的成本。

回想当初，斯托曼坚信软件厂商应该通过售后技术服务而不是专利权来谋取利润。为此，他与其他人通过自身的努力创建了一个软件乌托邦。现在，所有的开源系统都与斯托曼等人的基础性工作有着千丝万缕的联系。斯托曼等人的愿望，有好几个都已实现。GNU/Linux（本书后面将只写作“Linux”）的发展催生出了许多通过销售定制的Linux发行版本、并向Linux提供技术支持服务而取得成功（并且赢利）的公司，Red Hat和Slackware就是其中的代表。另一个例子是MySQL，它现在已是最为成功的开源数据库系统。

在当今世界，软件乌托邦能否从人们理想中的概念变成现实还是个未知数，但人们已经可以下载一整套系统软件和工具软件让一台个人电脑或商用电脑运转起来，而无需为软件本身付费。从操作系统到诸如数据库和Web服务器之类的服务器系统，再到办公软件，几乎所有门类的软件都有相应的免费版本可供任何人下载和使用。

1.1.1 为什么要使用开源软件

人们总会问到这样一个问题：为什么说使用开源软件是一个好主意？如果你想一劳永逸地堵住那些商业专利软件支持者的嘴，就必须为这个问题准备一个无可辩驳的答案。选用开源软件最重要的理由包括以下几点。

- 开源软件只需支付很少的费用，甚至无需支付任何费用就可以使用。这对非营利性的团体、大学和公共机构尤其重要——它们的预算总是在缩减，每年都需要用尽可能少的钱办尽可能多的事。
- 开源软件允许你根据自己的特定需求对之进行修改。
- 开源软件所遵守的许可证制度比商业软件许可证更加灵活。
- 开源软件往往比同类的商业专利软件更健壮（或者说经受过更多的测试）。
- 开源软件往往比同类的商业专利软件更加可靠和安全。

在决定选用开源软件的时候，强制或要求你给出以上这些理由的情况往往不多见，更常见的情况是你遭到反对。我的意思是，商业专利软件的支持者（或者说开源运动的反对者）会驳斥这些言论，发表有关“为什么你不应当在开发中使用开源软件”的言论。我们先从商业专利软件的立场去看看不使用开源软件的一些较常见的理由，然后再从开源软件的立场去驳斥他们。

1. 论点1：商业专利软件可以激发更大的创造力

这一结论的主要论据是：绝大多数企业级的商业专利软件都提供API（Application Programming Interface，应用编程接口）以便开发人员对其功能进行扩展，使其更加灵活且更便于开发人员发挥创造力。

这句话有一部分是对的。API确实能方便开发人员对软件进行扩展，但它们往往也会阻止程序员给原始软件添加新的功能。这类API通常会把开发人员限制在一个沙箱环境里，这进一步限制了开发人员的创造力。

注解 创建沙箱往往是为了限制程序员影响核心系统的能力。这么做的主要理由与安全性有关。API越开放，心怀叵测的开发人员就越有可能利用自己编写的恶意代码破坏系统或其中的数据。

开源软件也支持并提供API，但更重要的是开源软件让开发人员能够直接查看核心系统源代码。事实上，他们不仅可以看到源代码，更可以自由地修改它（这在开源阵营里是一种受到鼓励的行为）！只要它不具备你需要的重要特性，或者你需要系统能够读写某种特定的格式，你就可以亲自动手去修改核心系统。从这一点看，开源软件要比商业专利软件更能激发开发人员的创造力。

2. 论点2：商业专利软件比开源软件更安全

这一结论的主要论据是：在当今这个以因特网为纽带紧密联系的社会里，企业在信息系统安全性方面的要求要比以往任何时候都迫切。商业专利软件生来就更加安全，因为销售这些软件的公司已经投入了较大的力量去保证自己的产品可以经受住数字侵略者的攻击。

尽管这句话很可能被贴在商业软件公司会议室的墙上，作为公司的口号，但这个目标的实现情况不见得像这些公司的广告里所吹嘘的那么好。就拿微软公司的服务器版Windows操作系统来说吧。有关统计数字表明，Windows操作系统的服务器版本在安全性方面比不上Linux。虽然微软已经建立了一个成功而高效的补丁系统来保证Windows用户免遭已知攻击手段的伤害，但为Windows打补丁已成为服务器日常维护工作的一部分，这一事实已足以让我们怀疑微软产品的安全性达不到可以让用户免遭攻击的水平。（有些人人为此给出了这样一个说法：只要微软存在，就会有数字侵略者。）

Linux比Windows更安全的主要原因是参与Linux开发活动的开发人员遍布世界各地，他们一起努力让这个系统不会受到恶意攻击的伤害（术语称之为加固）。也就是说，来自世界各地的开发人员在共同加固着Linux操作系统。参与解决某个问题的开发人员越多，解决这个问题的办法就会越高明。这样一来，在Linux里发现的新漏洞会被非常迅速地堵上，从而把数字侵略者拒之门外。

与Linux的情况相比，微软公司负责加固Windows的程序员人数肯定要少得多，解决问题的思路当然也就少得多。这样一来，对Windows进行加固所需要的时间就会比Linux长得多。虽说不是所有的开源软件都能像Linux这样享受到这种“优待”，但它确实表明开源系统有能力从容地面对各种数字化威胁，并比同类商业专利软件更加安全。

3. 论点3：商业专利软件比开源软件经受过更多的测试

这一结论的主要论据是：软件公司卖的就是软件，而它们卖出的产品必须维持在一个高质量的水平才能吸引顾客来购买。开源软件没有这方面的压力，因而其测试环节不会像商业专利软件那么严格。

应该承认，这个论据非常有说服力。事实上，它说到了每一位信息技术采购人员的心坎里——这些人总认为花钱买来的东西要比不花钱就能得到的软件更加可靠且缺陷更少。可惜，这些人没有注意

到开源软件的一个重要概念。

开源软件是由来自世界各地的开发人员共同开发的，其中有许多人在扮演着“缺陷侦探”（测试人员）的角色，他们会为自己发现和报告的每一个缺陷感到自豪。有时候，开源软件公司会向发现bug的开发人员提供一些奖励。事实上，MySQL AB公司为那些在MySQL数据库系统里发现bug的开发人员支付的奖励相当可观。在我撰写本书的时候，任何人在MySQL AB公司的软件产品里发现bug，都可以从该公司获得一台苹果公司的iPod nano音乐播放器作为奖励。别说我没有告诉过你哟！

软件公司会招聘一些软件测试人员是事实，这些人也应该是他们这行里的佼佼者，但商业专利软件几乎都有一个既紧迫又不可更改的交付期限。设置这类期限的理由通常是为了保证重大战略版本的发布时机，或者是因为市场竞争的需要。在许多时候，这类期限会迫使软件厂商放松甚至放弃软件开发流程中的某些环节，而那往往会是整个流程中的最后一个环节——测试。稍微思考一下就会知道，测试人员接触软件的时间（测试时间）越少，他们能够找出的缺陷也就越少。

开源软件公司可以从世界各地的开发人员那里获得帮助和支持，而这意味着会有更多的人对它们的软件做更频繁的测试——换句话说，开源软件的测试强度要高于商业专利软件。

4. 论点4：商业专利软件比开源软件具备更多元的特性和更完备的功能

这一结论的主要论据是：商业专利数据库系统都是些非常完善和复杂的服务器系统，而开源系统的规模和复杂程度难以承担企业级海量数据的处理工作。

这句话对于一些模仿同类商业系统模仿得还不错的开源系统来说是正确的，但这句话绝不适用于像MySQL这样的数据库系统。与那些商业专利数据库系统相比，MySQL的早期版本是缺少了一些功能，但从5.0版开始，MySQL已经具备了在商业专利数据库系统里能够找到的所有高级功能了。

不仅如此，已经有许多事实可以证明，MySQL能够提供大企业在处理关键数据时所要求的可靠性、高性能和可扩展性。如果你想找一个典型，来证明开源系统也能像最好的商业专利数据库系统那样向用户提供全套功能的话，MySQL会是一个最佳的例子。

5. 论点5：商业专利软件公司的售后服务更好——因为它们有专人负责

这一结论的主要论据是：在购买软件系统的时候，卖方都会承诺该软件的开发商会帮助用户解决与该软件有关的问题；而开源系统根本不被谁所“拥有”，所以在遇到困难的时候很难找到帮手。

绝大多数开源软件都是由散布在世界各地的开发人员合作完成的。但目前的创业潮流是在坚持开源精神的基础上创办一家公司来有偿提供有关软件的技术支持和服务。事实上，绝大多数重要的开源产品都是在这种模式下推广的。比如说，MySQL AB公司拥有其MySQL产品的源代码。（MySQL的开源许可证的完整内容可以在网页www.mysql.com/company/legal/licensing/opensource-license.htm上查到。）MySQL AB公司对外提供多种技术支持服务，其中包括一周全天候服务、最快30分钟响应等。

开源软件的开发人员对疑难问题作出响应的速度往往要比商业软件的开发人员快很多。事实上，想直接与某个商业软件的开发人员对话几乎是不可能的事情。以微软公司为例，虽然该公司建立的规模庞大的技术支持机制能够满足几乎所有用户的需求，但如果你想与某位微软产品的开发人员直接沟通，必须先找对门路才行，这意味着你必须不厌其烦地与该公司各级技术支持部门联系，即使这样也不能保证一定能找到你想找的那位开发人员。

开源软件的开发人员以因特网作为他们的基本联络方式。因为他们几乎随时都在网上，所以他们很容易看到你在某个论坛或新闻组里发布的求助帖子。此外，像MySQL AB这样的开源公司还会主动地巡视自己的用户论坛，迅速对用户遇到的问题作出响应。

因此，购买商业专利软件并不一定能保证你会获得比开源软件更快的服务响应。在许多时候，与开源软件的开发人员取得联系要比与商业软件的开发人员取得联系更容易、更迅速。

6. 如果他们需要证据怎么办

我在上面列出了一些在为自己的企业选择开源软件时有可能会听到的反对声音。有些研究人员曾经试图证明上面那些说法是有根据的。有位名叫James W. Paulson的研究人员曾经对开源软件和商业专利软件（他称后者为“封锁源代码软件”）进行过实际体验的对比。针对上面列出的那几个论点进行了调查和体验之后，Paulson得出了这样一个结论：开源软件的开发模式明显优于商业专利软件的开发模式。Paulson的文章发表在2004年4月的*IEEE Transaction on Software Engineering*杂志上，文章的题目是“An Empirical Study of Open-Source and Closed-Source Software Products”。

1.1.2 开源软件是否对商业软件构成真正的威胁

不久以前，人们还一直认为开源软件不会对商业专利软件巨头构成威胁，但MySQL AB公司的两个最大的商业竞争对手正开始发出竞争威胁信号。微软公司继续发表着反对开源软件的言论，但在否认有危机感的同时，也开始承认MySQL是一个世界级的数据库服务器产品。而Oracle公司则采取了一种相当不同的手段。

Oracle公司最近刚完成了几笔企业兼并交易，它收购了开源公司SleepyCat和Innobase。这两家公司提供的解决方案都是MySQL系统的组成部分。虽然MySQL与这两家公司有技术支持合同在先，不会立刻因为它们被Oracle收购而受到影响，但业内观察人士普遍认为，虽然Oracle公司的表态没有火药味，但这家数据库巨头显然在准备与开源数据库阵营进行一场豪赌。数据库服务器市场2007年的预计交易额高达120亿美元，这场赌博争夺的就是利润和市场份额。

也许最能暴露Oracle公司野心的迹象莫过于它最近曾试图收购MySQL AB公司。还有什么能比最强劲的竞争对手想要占有你的东西更具有威胁性呢？MySQL AB公司的立场坚定，拒绝出售自己的劳动果实，这种行为非常值得赞赏。以上事件有两点值得关注：其一，开源的世界正在遭到蚕食；其二，MySQL AB公司的CEO们显然认为，继续坚持自己的信念做出世界上最好的数据库系统，就可以赢得更大的成功。

竞争压力并不仅限于MySQL与商业数据库系统之间。至少还有一种名为Apache Derby的开源数据库系统宣称自己可以替代MySQL，并在最近试图挤进“LAMP组合”并替代里面的“M”。Apache Derby的支持者提到了MySQL的许可证问题和功能限制。目前看来，MySQL的安装量似乎没有受到影响，这些“问题”也没有影响到MySQL日益流行的趋势。

什么是LAMP组合

LAMP代表着Linux、Apache、MySQL和PHP/Perl/Python，而所谓的LAMP环境是由这些开源阵营推出的服务器、服务和程序设计语言构成的软件开发/应用环境，它们可以让用户快速地完成一个高质量的Web应用程序的开发和部署工作。这个组合的核心组件如下。

- ❑ Linux，一种Unix风格的操作系统。Linux以高可靠性、高速度和适用于多种硬件平台而著称。
- ❑ Apache，一种Web应用程序服务器，以可靠性高和容易配置著称。Apache几乎能在所有的Unix操作系统变体上正常运行。

- MySQL, 众多Web应用程序开发者首选的数据库系统。MySQL以速度快和占用资源少著称。
- PHP/Perl/Python, 这是几种脚本语言, 用它们编写的程序片段可以嵌入HTML网页, 并由用户浏览事件激活运行。这几种脚本语言代表着LAMP组合里的主动性编程控制元素。它们用来联系系统资源和后端数据库系统, 一般向用户提供动态的内容。就这几种脚本语言而言, 虽说LAMP开发者大都比较喜欢使用PHP, 但它们其实都可以用来开发Web应用程序。

在LAMP组合下进行开发有很多好处。最大的好处是节约成本——LAMP组合的所有组件都可以通过免费的开源许可证获得。企业或个人在几个小时之内就可以完成这些软件的下载和安装工作, 并开始开发Web应用程序, 而软件方面的先期投入只需很少甚至根本不需要花钱。

向用户提供一个开源数据库系统的好处还可以从下面这件事情上看起来: 最近有几家商业数据库系统厂商也开始向用户提供免费版本。微软公司一边和开源阵营打着嘴仗, 一边却推出了SQL Server 2005数据库系统的免费版本SQL Server Express。虽然在你下载SQL Server Express时不收费, 微软公司也允许你随着自己的应用程序去传播它, 但仍然禁止以任何方式查看它的源代码或是对其进行修改。Oracle公司也为自己的数据库系统推出了一个名为Oracle Database Express Edition的免费版本。与微软一样, Oracle也允许你免费下载这个服务器软件, 并随着你的应用程序传播它, 但不允许修改或是查看它的源代码。这两种“免费”版本在功能上都有所裁减 (Oracle裁减得更多), 而且没有什么扩展的余地, 如果你需要的是一个完备的企业级数据库服务器, 还得另外掏钱购买额外的软件和服务。

显然, MySQL AB公司以及它的MySQL系列产品所显露出来的锋芒, 已经对商业数据库系统的巨头们构成了一定的威胁, 而那些商业软件的巨头们显然对这种威胁非常重视。不管Oracle近期收购那几家开源公司的真实目的是什么 (我们也许永远也不会知道), 它们都是在对来自MySQL AB公司的挑战作出反应。再看微软公司, 虽然它仍在试图让人们对于开源软件市场失去兴趣, 但显然它也明白推出免费软件的高明之处。

1.1.3 法律问题与《GNU 宣言》

商业专利软件的许可证是为了限制你的自由和限制你对软件的使用而设计的。绝大多数商业专利软件都明确地告知软件的购买者: 这个软件的所有权不属于你, 只允许你在非常特定的条件下使用这个软件。通常这意味着不允许以任何方式去复制、传播或修改。这些许可证都明确地写明: 源代码的所有权属于许可证的颁发者, 作为许可证接受者的你不得查看或改动源代码。

开源系统一般使用的是一种基于GNU的许可证协议, 它们通常都允许自由地使用原始的源代码, 但往往会加上一条限制要求你在修改源代码之后必须公开所做的修改, 或是要求你把有关代码发送给该软件的法定拥有者。此外, 大多数开源系统几乎都使用GPL协议, 它明确地写着其目的是为了保证人们有权对有关软件进行复制、传播和修改而不受任何约束。值得注意的是, GPL协议对你将如何使用有关软件没有做出任何限制。事实上, GPL专门保证了用户以自己所希望的任何方式使用软件的权利。GPL协议还确保你有权随意查看源代码。所有这些权利都明确地写在《GNU宣言》和GPL协议上 (www.gnu.org/licenses/gpl.html)。

最有趣的是, GPL专门允许在传播原始软件的时候可以收取一定的传播费 (介质费), 还允许为了创建衍生产品而使用完整的或修改过的系统。你的衍生产品同样会受到GPL的保护, 但前提是必须把你对于源代码的修改向所有想知道它的人公开。

这些限制并不禁止你用自己的劳动成果获得报酬。只要你通过有关软件的原始拥有者向人们公开自己的源代码，就可以靠你的衍生产品向顾客收取费用。有些人认为这意味着你永远也不会有真正的竞争优势，因为任何人都可以自由地获得你的源代码。这话似乎有几分道理，但实际情况却正好相反——Red Hat和MySQL AB公司都实现了盈利，它们的业务模型就建立在GPL的基础上。

在跃跃欲试之前，请仔细阅读一下GPL上的责任条款。这里提醒大家注意的是，GPL要求你为自己的衍生产品添加一个标志，把它的“血统”（原始软件及其许可证情况）写清楚。

责任条款对大家来说应该不是什么新鲜事，绝大多数的商业许可证上都有类似的东西。GPL的特别之处是免责条款：软件的原始作者和修改者（或传播者）对于安装或使用该软件所导致的损失和伤害不承担任何责任。这是因为斯托曼不想让那些唯恐天下不乱的律师们瞄着开源软件的可靠性去赚钱。道理很简单：既然软件是你免费获得的，用或不用的决定权在你自己手里，再让别人为它的性能或是因为使用它而蒙受的损失承担责任就没有道理。总之，如果接受GPL，就不能获得任何保证。

开源运动的反对者喜欢把GPL的免责条款当作一个不使用开源软件的理由，他们认为“风险自负”代表着会有很大的风险。这种想法可以理解，但你完全可以从开源公司购买技术支持服务来降低甚至消除这些风险。开源公司提供的技术支持服务通常都有某种程度的可靠性保证和保修期限，这或许是为开源软件购买技术支持服务的最佳理由。总之，如果选择购买技术支持服务，就能在可靠性方面获得某种程度的保证。

让你在软件的醒目位置放上一个标志的要求并不是那么难。GPL作出如此要求的目的是有两个：一是让最终用户了解这个软件的来龙去脉；二是向用户表明这个软件受GPL保护。这可以让使用这个软件的人确切地知道自己有权使用、复制、传播和修改这个软件。

在《GNU宣言》里，“如何获取GNU”部分大概是最重要的声明了。该宣言明确指出：虽然任何人都可以修改和再次传播GNU软件，但绝不允许任何人限制它的进一步传播。这意味着没人可以把一个基于《GNU宣言》的开源软件转变为一个专利系统，或者对它作出专利性的修改。

1. 归属权

对开源软件许可证问题的讨论如果不谈到归属权问题就不算完整。简单地说，归属权就是人们拥有某种东西的权利。人们平常所说的归属权都是些有形物体，即看得见、摸得着的东西的归属权。在软件行业，归属权这个概念就有些难以把握了。当我们说软件有归属权的时候，我们到底想要表达什么意思？归属权的概念是仅适用于源代码、二进制代码（可执行代码）、文档，还是包括所有这些东西？

在谈到开源软件的时候，归属权的概念往往是一个很难说清楚的话题。如果某个软件是由世界各地的开发者合作完成的，谁将是它的拥有者？绝大多数开源软件都是从某个人或某家公司已经开发到一定阶段的项目开始成形的，只有当这个软件成熟到对某些人有用的时候，它才会变成开源软件。这个软件是处于尚不成熟的初期阶段，还是已经具备了一定的可靠性并不重要，真正重要的是某人发起了这个项目。这个“某人”通常会被承认是这个软件的拥有者。就拿MySQL来说，因为是MySQL AB公司发起的这个项目，所以MySQL系统的拥有者就是该公司。

根据MySQL所遵守的GPL许可证，MySQL AB公司拥有其所有的源代码和其他人在GPL许可证下对它作出的任何修改。GPL许可证只给了你修改MySQL的权利，但它没有把宣称MySQL源代码归属权的权利授予你。

2. 道德问题

道德是一个谁都害怕讨论的话题，可就在你第一次使用开源软件的时候，道德问题出现了。比如说，开源软件可以免费下载，但你必须把对它作出的任何改进都报告给它的原始拥有者。你已经把东西送出去了，还靠什么去赚钱呢？

要想把这个问题弄明白，必须先把斯托曼起草GNU许可证时想要达到的目的搞清楚。他的目标是在全世界的程序员当中创办一个团结合作的集体。他希望软件的源代码能够让人们共享，开发出来的软件可以让任何人自由地使用。你用劳动成果去赚钱的权利并没有受到限制。你完全可以售卖你的衍生产品，只是不能声称拥有它的源代码而已——回报全球开发人员社区是你的道德义务（以及法定义务）。

与开源软件有关的另一个道德问题出现在你修改的源代码仅供自己使用的时候——公开还是不公开你的修改？比如说，你下载了MySQL的最新版本，并增加了一项功能以方便你使用自己爱用的SQL命令缩写，因为你对输入长长的SQL语句感到厌烦了（我敢肯定在什么地方已经有人这么做过）。

在这类情况下，你对系统的修改只对个人有好处，其他人未必会喜欢。那么还有必要公开吗？这种情况我们当中的绝大多数人都不会遇上，但如果你在这次修改之后一直使用那个软件，并且最终创建了出了一个衍生产品，问题可就来了。我的忠告是：无论因为什么理由，只要你打算修改源代码，就必须把后果考虑清楚。简单地说，只要你作出的修改能给工作带来方便和价值，不管扩展还是缩减了有关的功能，你都必须承认它的归属权是原始作者。

如果你对源代码进行的修改是课堂练习或学术研究性的（就像我在本书稍后的内容里做的那样），你应该在完成练习或实验后立刻撤销所有的改动。有些开源软件对这种练习或研究性质的使用也有规定，但绝大多数程序员会把对源代码进行研究和实验看作是一种“使用软件”，而不是“修改软件”的行为。一般来说，出于学术目的而改动的源代码可以不提交给原始作者。

1.1.4 将开源进行到底

自由精神是驱使理查德·斯托曼致力于改变软件开发格局的源动力。它是开源运动的催化剂，这场运动已经演变成了一场革命，而这场革命为众多企业提供了新的发展契机：投资于低成本的软件系统可以让企业在不减少利润的情况下降低产品价格以提高自己的市场竞争力，从而避免被竞争对手拉开距离。

把开源软件作为其产品线一部分的企业，应该说是这场革命中最大的受益者。这类公司通常会采用一种基于GPL许可证的商业模型，因为这可以让它们在坐享遍布世界各地的开源程序员的集体智慧和经验的同时，仍能通过自己的创意和附加劳动获利。

开源运动在软件行业既有支持者，也有反对者。有些人根本看不起开源运动，认为这是对商业专利软件行业的冲击，并断言开源是一种倒退，不可能长久。他们认为生产、传播或使用开源软件的组织只会昙花一现，世界迟早会重归秩序并把开源软件遗忘在脑后。还有些人对开源持悲观态度，他们认为开源软件没有任何盈利的可能性，不愿意在这种他们认为不会有结果的事情上多费脑筋。但更多的人则把开源软件视为可以把全人类从商业专利软件的专制下解放出来的大救星，认为这迟早会迫使那些软件业巨头把它们的专利产品转变为开源或某种与之类似的东西。孰是孰非还有待时间的检验。我个人认为开源产业是一个充满活力和机遇的产业，你可以在这里找到志同道合的朋友，一起为创建出安全、可靠和健壮的软件而努力奋斗。

不管你本人对此有何看法，开源运动已经在世界各地的软件开发者当中掀起了一场革命，这是一个不可否认的事实。现在，你应该对这场开源革命有了一个比较全面的了解，是否投身到这场革命中去要由你自己来决定。如果你的回答是“是”（我当然希望我已经说服了你），那么欢迎你加入程序员的全球合作阵线。

1.2 用 MySQL 进行开发

什么是开源软件，使用和开发一个开源软件时需要注意哪些事项，前面已经讲得很清楚了。接下来，我们将开始学习如何使用MySQL来开发新的软件产品。你们即将看到，MySQL向开发者提供了一个独一无二的机会，去探索一个大型服务器软件里的技术秘密，这种探索不要求你听命于别人，不限制你必须遵守哪些规则，也不限制你必须使用哪一种API。

MySQL的拥有者是MySQL AB公司。“AB”是瑞典语“aktiebolag”的字头缩写，意思是“股份公司”。从投资创业到推出一个世界级的开源关系数据库系统，MySQL AB公司的发展历程令人信服地为人们指出了一条与商业数据库市场迥然不同的道路。MySQL AB公司的盈利办法是销售各种商业许可证、技术支持服务和专业开发服务，其中包括咨询、培训和产品认证。

MySQL是一种最适合在客户端/服务器环境下使用的关系型数据库管理系统，但它也可以被用作一个嵌入式的数据库组件。当然，如果你曾使用过MySQL，就会知道它的能力并很可能会毫不犹豫地决定选用MySQL来满足你的部分或全部数据库需求。

MySQL已成为世界上最流行和最成功的开源数据库系统。这种流行在很大程度上要归功于它的可靠性、性能和易用性。目前，MySQL在全世界的装机量已经超过了800万套。MySQL AB公司之所以能取得如此巨大的成功，是因为它一直坚持着这样一个理念：“要让最先进的数据管理软件成为人人都能找得到、用得起的东西。”这个核心价值观已经通过MySQL AB公司的经营目标得到了贯彻，即要使其数据库产品可以做到：

- ☐ 成为全世界最优秀和最流行的数据库产品；
- ☐ 人人都找得到、用得起；
- ☐ 易于使用；
- ☐ 在保证速度和数据完整性的前提下，不断改进和完善；
- ☐ 容易扩展和进化并让这一过程充满乐趣；
- ☐ 没有缺陷。

显然，MySQL AB公司已经实现了所有这些目标，并继续以产品的质量和性能让世界各地的数据库专业人士感到惊讶。

MySQL的诞生过程和它的内部结构对很多人来说还是个秘密。在这个系统的最低层有一个按照多线程模型用C和C++语言开发出来的服务器，这个核心组件的主要部分是在20世纪80年代早期建立起来的，后来又在1995年用一个SQL层对其进行了一些修改。MySQL是用GCC编译的。GCC编译器可以让人们根据目标环境对编译过程做出非常灵活的调整，这意味着MySQL经过编译能在几乎所有类型的Linux操作系统上运行。MySQL AB公司在把其产品移植到Microsoft Windows和Macintosh操作系统方面，也取得了令人瞩目的成功。出于可移植性和速度方面的考虑，MySQL的客户工具大部分是用C语言编写的。适用于.NET、Java、ODBC等环境的客户端库和访问机制也都应有尽有。

为了在规划和开发软件新版本的同时还能保证现有产品得到进一步完善,MySQL的开发活动采用了平行开发策略,也就是把软件的整个开发过程划分为一系列阶段,在每个阶段都会形成多个版本。MySQL的开发过程可以划分为以下几个阶段:

- 开发——完成新产品或新功能的规划设计并开始把它作为整个开发树的一个新的分支加以实现。
- α ——进行功能细化和缺陷纠正(调试纠错)。
- β ——“冻结”功能部分(不再添加新的功能),继续进行高强度测试和缺陷纠正。
- γ ——简单地说,这是产品正式发布之前的最后一个阶段,代码不会再有大的改动,但还需要对它进行最后几轮测试。
- 稳定——如果在 γ 阶段的最后几轮测试中没有发现重大的缺陷,有关代码将被宣布为稳定的,产品发布已指日可待。

你们可能已经注意到,MySQL软件有各种各样的版本,其原因是它们分别来自上述这些阶段。平行开发策略使得MySQL AB公司得以同时兼顾新产品/新功能的开发与现有版本的完善,因而5.1版里的新增功能稍后又出现在了4.0.10版里并不是什么稀奇的事情。应该承认,这种事很容易把人搞糊涂,毕竟商业专利软件已经让我们当中的许多人形成了“版本越高、功能越新”的思维定势。我们可以根据MySQL的版本编号来区分它们,这个编号的前两个数字代表着产品系列,最后一组数字代表着该产品是那个系列的第几个版本。比如说,5.0.12是5.0产品线的第12个版本。

提示 向MySQL AB公司反映问题的时候,一定要把完整的版本编号说清楚。只说“ α 版”或“最新版”还不够明确,会耽误问题的解决。

这种同时开发并推出多种版本的策略有一些很有意思的副作用。在某个企业看到仍使用着一个相当老的MySQL版本并不是什么新鲜事。事实上,我曾工作过的几个单位到现在用的还是4.x系列的版本。这种策略给用户带来的好处是用不着密切关注系统升级问题。商业专利软件可做不到这一点,每当它们发布一个新的版本,就会结束对老版本的开发,有时甚至连对老版本的技术支持服务也终止了。每当商业专利软件在体系结构方面做出重大调整,用户都不得不亦步亦趋地对自己的系统环境和开发工作做出相应的改变。如果你的产品线建立在商业专利软件的基础上,这种被动的改变往往会让你的产品线维护成本大大增加。同时开发并推出多种版本的策略,可以让企业摆脱这种负担,因为这样能够继续享受到技术支持服务,其产品的生命期就可以更长。即使在体系结构方面发生了重大改变——比如MySQL 5.0推出的时候,企业也有足够的时间以最有效率的方式去协调自己的资源,而不必急于改变自己的长期计划。

MySQL的任何版本都可以免费下载,但你首先应该考虑清楚你想用这个软件来做什么。如果你的计划是把这个软件用在日常工作环境里的一台企业级服务器上,就应该只下载某个MySQL产品线上的稳定版本。另一方面,如果你是在建立一个基于LAMP环境的新系统,或是想换一种开发环境,在其他开发阶段所推出的MySQL版本应该也可以考虑。绝大多数人都会下载最新产品线里最稳定的版本建立自己的环境。因为本书对MySQL软件的使用主要是做练习和做实验,所以MySQL的任何版本(阶段)都可以用。

MySQL AB公司向从事开创性开发工作的程序员们推荐最新的 α 系列。它们这么推荐的含义是:

如果你打算给MySQL增加新的功能，并希望成为程序员全球合作阵线中的一分子，那么应该在 α 阶段来做这件事——你的代码将接受最多的测试（向全世界公开），最起码也应该在最后的 γ 阶段（产品发布）之前。还有一点需要注意：虽然某个版本的各开发阶段是新功能是否稳定可靠的标志，但千万不要想当然地认为，较低的阶段就意味着不稳定、不可靠，较高的阶段就意味着稳定可靠。根据你所使用的这个软件的具体用途，其稳定性可能会有所差异。比如说，如果你正在使用MySQL开发一个LAMP环境下的电子商务网站，并且没有用到在开发阶段或 α 阶段引入的任何新功能，这种用途下的稳定性与你具体使用的MySQL版本来自哪个阶段是没有什么关系的，它们应该有同样的表现。作为一般用户，如果你懒得动脑筋，这里有一个简单的规则：先看哪条产品线能够提供你所需要的功能，再把这条产品线里目前处于最高阶段的那个版本（版本号的最后一个数字越大越好）找出来，就是它了。

1.2.1 为什么修改 MySQL

修改MySQL可不是小事情。如果你是一位有经验的C/C++程序员并熟悉关系数据库系统的内部结构，你可以直接动手。如果不是这样，还是先花点儿时间去考虑一下为什么要修改一个数据库服务器系统，再制定一份详细的修改计划比较稳妥。

需要自己动手修改MySQL的理由很多。比如说，你需要一种新的数据库服务器或客户功能，但没有现成的东西能提供这种功能。再比如说，你有一套定制的应用程序需要一种特定类型的数据库行为，与选用一种商业专利系统相比，自己动手修改MySQL来满足你的要求更简单、更合算。通常你所在的组织无力自行开发一个像MySQL那样复杂精细的数据库系统，但你的解决方案又需要有个东西作为基础。如果你想让自己的应用程序达到世界一流的水平，还有什么办法会比使用一个世界一流的数据库系统作为它的基础更好呢？

注解 开源的魅力在于：如果某项功能真的非常有用并且有人认为它可以赚钱，这项功能就会以其独到的方式融入某个产品。肯定会有某人在某地研究和建立这项功能。

和所有讲求效率的程序员一样，你首先要为自己将要做的事情制定一个计划。从你最熟悉的设备和资料开始，把你认为需要数据库服务器（或客户端）去做的所有事情列成一份清单。抽出时间去评估一下MySQL，看看你需要的功能是否已经存在并记录一下它们的行为特点。在完成这项研究工作之后，你将对自己要做些什么有一个更清晰的认识。这项分析可以让你知道需要对哪些功能做哪些修改（最好把它们也列成一份清单）。一旦确定了需要添加的功能，就可以去查阅MySQL的源代码并开始尝试增加那些新功能了。

警告 在制订修改计划时，一定要对MySQL的现有功能做全面彻底的评估。你应该把与你的需求相类似的所有SQL命令都试过来。就算MySQL的现有功能都帮不上忙，对它们的能力进行评估，也会让你得到一条能与你将要添加的新功能进行对照的基准线（行为特征、性能等）。可以确定的是，全球开发人员社区会对每一个新功能进行审查，如果他们觉得某个“新”功能比不上现有的某个功能，就会废弃这项“新”功能。

学习MySQL源代码的最佳办法是反复阅读！本书向大家介绍了MySQL的源代码，讲授了如何添

加各种新功能，并介绍了与需要修改哪些代码（以及不应该修改哪些代码）有关的最佳实践经验。接下来的几章还将详细地告诉大家如何获得MySQL源代码，以及如何把修改合并到一个适当的代码路径（分支）里去。还将介绍MySQL AB公司编程指南的细节，对代码样式以及应该避免哪些代码结构等问题提出了明确的建议。

1.2.2 MySQL里哪些可以修改，有什么限制

开源软件的魅力在于你可以去查看它们的源代码（这一权利受开源许可证的保护）。这意味着你可以了解到整个软件的内部机制。想知道MySQL里的优化器是如何工作的吗？只要把它的源代码下载下来，并肯花时间去钻研，你就可以把这个问题搞清楚。

放大到整个MySQL软件，事情可就没有这么简单了。MySQL的源代码很复杂，难以阅读和理解。有人说那些代码的条理很不清晰，也有人说那是天才头脑才能想出的东西。总而言之，阅读那些源代码对最优秀的C/C++程序员来说也是一个挑战。

那些复杂的C/C++代码有些让人望而生畏，但那纯属技术问题，与你是否有权修改这个软件没有任何关系。绝大多数程序员修改MySQL源代码的目的是为了增加几条新的SQL命令，或是把某个现有的SQL命令改得更能满足自己对数据库的需求。请不要满足于此，MySQL里可以修改的东西并不仅限于其SQL行为，优化器、查询命令的内部编码格式，甚至是查询缓存机制都可以修改。

自行修改MySQL源代码的做法很少会遇到来自程序员方面的质疑，你最可能遭遇的阻力往往来自你的高级技术主管。比如说，我最近对MySQL源代码进行的一次修改，就在我的高级技术主管那里卡了壳，原因是我修改了服务器部分的重要代码。有位主管说我的修改“完全抛开了30年来的数据库理论和屡试不爽的实践”。我当然希望你们永远也不会遇到类似的情况，可有些事情是无法回避的。在遭到质疑的时候，如果你已经仔细研究过MySQL的现有功能和它们为什么不能满足（或者只是部分地满足）你的使用要求，就应该用无可争辩的事实来回答对方。如果质疑来自你的高级技术主管，别忘了提醒他们开源软件的传统美德：允许别人修改，总是有人在修改。当然，你还应该向每个人解释清楚你添加的新功能都有哪些用途，以及它将如何改善整个系统。如果你能把这几件事情都做好，平息这场质疑风暴应该没什么问题。

与修改MySQL有关的另一个常见问题是：“为什么选用MySQL？”专家们会立刻指出有好几种开源数据库系统可供选择。最流行的是MySQL、Firebird、PostgreSQL和Berkeley DB。选用MySQL而不是其他数据库系统来开发某个项目的理由主要有以下几点。

- ❑ MySQL是一个支持全套SQL命令的关系数据库管理系统。有些开源数据库系统（如PostgreSQL）是对象关系数据库系统，它们需要通过一个API或库进行访问，而不是接受SQL命令。有些开源系统采用的体系结构可能不适用于你的环境。比如说，Apache Derby是基于Java的，在嵌入式应用里发挥不出最佳性能。
- ❑ MySQL是用C/C++语言编写的，在几乎所有的Linux平台以及Microsoft Windows和Macintosh OS上都可以编译。有些开源系统不一定支持你选择的开发语言，如果你开发出来的系统最终必须运行在某个版本的Linux平台上，就不能不考虑这个问题。
- ❑ MySQL采用的是客户端/服务器体系结构。有些开源系统只是一个基于客户端的嵌入式系统，没有多少可扩展性。比如说，Berkeley DB是一套客户端库，不是真正意义上的数据库系统。
- ❑ MySQL是一种成熟的数据库服务器，其稳定性已在实践中得到了证明。有些开源数据库系统

没有MySQL那么大的装机量，难以保证在一个企业级数据库服务器里提供你所需要的功能。

很明显，各种挑战只会来自开发需求和进行修改时的环境。有了MySQL，可以肯定的是，你完全可以访问到所有的源代码，进行修改所面临的限制只有你的想象力。

1.2.3 MySQL 的许可证问题

MySQL是一个采用GPL发行的开源软件，它的服务器和客户端软件以及各种工具和库都在GPL的效力范围内。MySQL AB公司早已把GPL作为其商业模型中的重中之重。他们是GNU开源阵营的中坚力量。不仅如此，MySQL AB公司还要求与之签约的风险投资商都必须理解和接受同样的理念和许可证。

MySQL AB公司向全球开发人员社区公开自己的源代码后获得了许多好处：有遍布世界各地的程序员定期对这些源代码进行评估，有第三方组织定期对这些源代码进行审计，开发工作促成了进行开放的交流与反馈活动和反馈的论坛，源代码在许多不同的环境下接受编译和测试。除了MySQL，没有一家数据库厂商能够在做到这些的同时保持世界一流的稳定性、可靠性和特色。

MySQL也有采用商业许可证发行的产品。MySQL AB公司在其商业许可证里申明了自己对源代码（如前所述）以及名称、商标图案和文档（如书籍）的拥有权。这在开源公司当中是独一无二的，因为绝大多数开源公司都不申明拥有任何东西；而他们则拥有他们的经验和技术的知识产权。MySQL AB公司在支持开源运动和软件事业发展的同时，还保留了软件的知识产权。也许你们还不知道，MySQL AB公司有自己的开发团队，该团队的100多位成员遍布世界各地。参与开发MySQL的程序员来自世界各地，MySQL AB公司聘用了他们当中的许多人。

有些人认为MySQL AB公司的做法违背了斯托曼和FSF的初衷，这种看法是不对的。MySQL AB公司是围绕开源数据库系统而创办的一个企业，这家企业在坚持开源精神的同时保留了聘用本行业成员的权利。MySQL AB公司的成功向世人表明，在公开自己创意的同时还通过销售它们来赚钱是完全有可能的。

应该承认，这种双重许可证概念造成了一些混乱。比如说，你在什么情况下才可以用GPL去对抗商业许可证呢？GPL最适用于软件的正常使用、与其他程序员合作添加或优化软件的功能和从事学术实验等场合。商业许可证则最适用于需要质量保证/技术支持服务，或是在一个非常关键的应用程序任务中使用有关软件的场合。

在修改软件时应该遵守哪个许可证也是一个容易引起混乱的话题。如果你添加的新功能不止你自己的用户感兴趣，你应该考虑使用GPL并把改动报告给MySQL AB公司。虽然这意味着你放弃了自己拥有那些改动的权利，但将获得的是世界一流的技术支持和MySQL系统的所有其他优势。如果你做的修改只对自己有用，并且也不打算把自己的修改（以任何形式）弄成一个产品去传播发行，那么使用哪个许可证都没有关系。

如果你使用了GPL但没有公开做出的修改，你做的那些修改将得不到任何支持，维护它们的责任将完全由自己一个人来承担。这在你打算升级到MySQL的一个新版本的时候可能会引起一些问题，至少你必须把以前做过的所有修改再重复一遍。重新修改未必是件难事，但却是一件需要花费精力去仔细计划的事情。MySQL AB公司为使用GPL许可证的用户准备了许多种技术服务套餐。MySQL的官方技术支持网站（www.mysql.com/support/community_support.html）上有许多链接方便用户订阅免费的邮件表，加入论坛和提交缺陷报告。如果你愿意交纳一些费用，还可以享受到专家咨询和培训服务。

如果你使用的是商业许可证，就可以从MySQL AB公司购买技术支持服务来帮助自己完成必要的修改。你甚至可以买到允许你拥有那些改动的权利——如果你打算由自己来打包并发行有关的源代码，这个权利将非常重要。表1-1汇总了MySQL AB公司目前所提供的各种技术支持套餐。不管你使用的是哪一种许可证，你都可以享受这些被统称为MySQL Network的技术支持服务，但使用GPL的用户会受到少许限制。

表1-1 MySQL Network支持选项

服务项目	普通用户	银卡用户	金卡用户	白金用户
软件维护和升级	是	是	是	是
服务向导	是	是	是	是
免费使用知识库	是	是	是	是
故障报告	2次	无限次	无限次	无限次
电话支持		8×5(周一到周五)	24×7	24×7
上门服务到位时间(最大)	2个工作日	4小时	2小时	30分钟
紧急响应时间(最大)			30分钟	30分钟
远程故障排除			是	是
模式审查				是
对查询命令进行审查				是
性能调整				是
代码审查(客户端程序)				是
代码审查(用户自定义函数)				是
代码审查(服务器端程序)				是
账户管理专员				可选
损失赔偿			可选	可选

注解 上表中的“损失赔偿”仅限于与MySQL有关的版权和专利侵权纠纷。

1.2.4 到底能否修改 MySQL

究竟能不能修改MySQL？经过上面这些关于在GNU公共许可证下使用开源软件的限制的讨论之后，你也许会这样问。答案很简单——你可以修改！

可以修改采用GPL许可证发行的MySQL，但如果打算传播所做的修改，就必须把那些修改无偿地提交给项目的拥有者，你本人只是尽到了全球程序员合作阵线一员的义务。如果修改软件的目的是科研或教学，你可以不公开修改。要不要公开你的成果关键要看它到底有多大的用处。如果添加的新功能对其他人也有用，就应该奉献出来让大家共享。

也可以修改采用商业许可证发行的MySQL，但仅限于两种情况：要么那些修改今后只能用在你自己的内部开发工作中，不允许再发行；要么你修改后的MySQL必须成为你推出的某个商业产品的一部分，仍按商业许可证发行。

不管你选择的是哪一种许可证，修改这个系统的决定权由你掌握。

1.2.5 修改 MySQL 的指导原则

在修改类似MySQL这样的大系统时，一定要谨慎行事。关系数据库系统是由一系列用来快速可靠地存取各种数据的服务构成的复合集合。很少有人会冒冒失失地打开源代码，把自己的代码添加进去，然后看到底会发生什么事（但还是可以尝试一下看看结果如何）。你应该先制定一个周密的修改计划，并准确地找出与你的需求有关的那部分源代码。

我曾对MySQL这样的大系统进行过修改，所以我想以一个过来人的身份在这里给出几条简单的建议，帮助大家能够顺利完成修改MySQL的工作。

你应该做的第一件事是决定使用哪一种许可证。如果你使用的MySQL版本是按开源许可证发行的，而你本人又具备自行完成那些修改的能力，就应该继续使用GPL许可证。在这种情况下，你应该按照开源的传统，把你的劳动成果回馈给向你无偿提供各种资源和帮助的人们。根据GPL许可证的条款，开发者有义务公开他所做的修改。如果你使用的MySQL版本是按商业许可证发行的，或是需要别人为你做出的修改提供技术支持服务，那么你应该购买相应的MySQL Network技术支持服务，并就你的修改与MySQL AB公司取得联系。如果你既不打算传播做出的修改，又有能力在未来的MySQL版本中支持它们，就不需要改动你的许可证。

我的下一条建议是建立一个开发日志，把你做过的每一处修改和你认为有意思的每一个发现记录下来。这是每一步工作的记录，更可以作为文档帮助你在日后回忆起曾经做过的工作。以往的工作记录经常能给人带来一些意想不到的启发。我就经常在我的工程笔记本里发现一些金点子。

在修改源代码的时候，还应该在源代码里加入一些注释。可以在你添加或修改的代码段的前后用一些注释语句把你的想法写下来。在以后使用文本编辑器或文本搜索程序查找你做过的修改时，这些注释会为你提供许多便利。下面这个例子演示了一种给修改加上注释的具体办法。

```
/* BEGIN MY MODIFICATION */
/* Purpose of modification: experimentation. */
/* Modified by: Chuck */
/* Date modified: 3/19/2006 */
if (something_interesting_happens)
{
    do_something_cool;
}
/* END MY MODIFICATION */
```

最后，要敢于和善于利用MySQL官方网站上免费开放的知识库和论坛，或是向全世界的程序员寻求帮助，这些都是你的宝贵财富。不过，在论坛上发布求助帖子之前，一定要先做好准备工作。没有什么比在某个论坛上发出求助帖子之后，立刻有人回复说“去查看随机文档的某某小节”更容易让人尴尬的事情了。你的求助帖子应该言简意赅。不必罗列一大堆理由去解释你为什么要做正在做的事情，只需把你遇到的问题和手里与这个问题有关的所有信息写清楚就行了。还要注意的，一定要把求助帖子发布到正确的论坛上。绝大多数论坛都有专人（俗称“版主”）在维护，如果你有什么不明白的地方，可以与版主联系以确保你的帖子发布在正确的论坛里。

1.3 实际的例子：TiVo

有没有想过是什么让你的TiVo保持运转的？如果我告诉你它运行在某个嵌入式版本的Linux上，你会感到惊讶吗？

Jim Barton和Mike Ramsay在1997年设计出了最早的TiVo。它最初只是一个基于家庭网络的、向瘦客户端提供流媒体内容的多媒体服务器。显然，像TiVo这样的设备不仅要易学，更要易用，而最重要的是，它必须能够实现无错误的运行，并能处理好电源故障（以及用户的人为失误）。

Barton有着丰富的Linux使用和开发经验，他在SGI公司工作期间曾负责过把Linux移植到SGI Indy平台的项目。因为对Linux稳定的文件系统、网络、内存处理和开发工具支持印象深刻，Barton相信把Linux的某个版本移植到TiVo上是可行的，他认为Linux的性能能够满足TiVo产品在数据实时处理方面的要求。

Barton和Ramsay的设想受到了同仁们的质疑。那时候，人们大都对开源持怀疑和讽刺态度，当时的某些商业软件专家曾断言开源软件的可靠性永远也达不到实时环境的要求。此外，他们认为把一个商业专利产品建立在GPL的基础上会导致今后无法进行修改——如果他们想进行改进，这个项目就会演变成一场版权诉讼的噩梦和无休止的法律争端。Barton和Ramsay并没有被吓倒，他们仔细研究了GPL并得出了这样一个结论：GPL不仅完全站得住脚，还可以让他们保护自己的知识产权。

Barton和Ramsay的最初设想是把TiVo搞成一个服务器产品，但他们很快发现，当时的带宽不足以支持这么大的应用。于是，他们重新设计了产品，把它改成了一个有点儿像一台比较复杂的录像机的客户端设备，并给它取名为TiVo Client Device (TCD)。他们打算建立一个提供电视节目指南的收费服务，而用来接收和处理有关信息的设备就是TCD。这将使家庭用户能够提前选择他们想看的节目并控制TCD到时候把那些节目录下来。简单地说，他们设想中的产品和现在的数字录像机(DVR)差不多。

TCD设备的硬件包括一台带有硬盘和内存的微型嵌入式计算机，硬件接口部分使用一个MPEG-2编码/解码器来读写视频信息（视频输入和视频输出）。其他的输入/输出(I/O)设备包括音频和远程通信（用来访问TiVo服务）。TCD还必须具备多处理能力，这样它才能一边录制着一个信号（频道）一边播放另一个（频道）。这些功能需要有一个良好的内存和硬盘管理子系统。Barton和Ramsay认为这些目标对任何控制系统来说都是一个挑战。不仅如此，视频接口部分还必须做到永远也不会掉线或崩溃。

Barton和Ramsay最需要的是一个符合以下要求的系统：有稳定可靠的硬盘子系统，支持多任务，具备硬件（CPU、内存）优化能力。Linux因此而成为TCD设备的操作系统的当然选择。量产目标和预算上限限制了CPU的选择。IBM公司生产的PowerPC 403GCX被选为TCD设备的CPU，但可惜的是没有能在这个处理器上运行的Linux版本。这意味着Barton和Ramsay必须把Linux移植到这个处理器平台上来。

移植工作很成功，但Barton和Ramsay发现他们需要对Linux内核进行一些特殊的定制，才能满足硬件的要求和限制。比如说，为了加快对录制/播放的视频信号的处理速度，他们让数据不再经过Linux文件子系统读写缓冲区的缓存。他们还增加了大量的性能优化功能、操作日志功能和故障自动恢复功能，以确保TCD可以从断电或用户误操作中迅速恢复过来。

TCD设备的应用程序是在一台Linux个人电脑上完成后，再移植到已经改好的Linux操作系统上去的。整个过程非常顺利，这证明了Linux操作系统的稳定性和兼容性。在完成了操作系统的移植和应用程序的开发工作之后，Barton和Ramsay进行了大量的测试，并终于在1999年3月推出了世界上第一

台DVR机。

现在，在内部装有定制的嵌入式Linux操作系统的各种设备当中，TCD已是销量最大的消费产品之一。TCD的故事是通过修改开源软件而取得巨大成功的一个光辉例子。这个故事还有一个尾声：Barton和Ramsay公开了他们修改Linux内核时编写的全部源代码，他们开发出来的一些新功能已经出现在了Linux内核的最新版本里。

说服你的老板修改开源软件

如果你有一个好创意和一个基本的商业模型，开源这条道路可以大大加快你的产品进入市场的时间。事实上，你的项目很可能会节省下大量的开发经费，并让你的产品先于竞争对手进入市场。与从零开始开发一个软件相比，如果只需对一些现有的开源软件做些修改就可以形成产品的话，时间和金钱方面的节约效果就更显著了。当然，前提是你已经做好了前期调研并能拿出令人信服的关于使用开源软件的成本效益报告。

不幸的是，很多经理已经被商业专利软件公司弄得团团转，他们往往会不假思考地拒绝以开源软件作为产品的基础。怎样才能改变他们的想法呢？先给他们讲讲TiVo的故事，再破除那些关于商业专利软件的不实观点，然后把GPL许可证的好处和开源软件的可靠性向他们讲解清楚。在谈话过程中不要表现得过于激动，你的热情说不定会被高级技术职员视为一种要挟。

列出一份名单，找出站在商业专利软件立场上的所有技术主管，然后找机会与他们好好讨论一下开源软件并回答他们的问题。保持耐心在此时是最重要的。技术主管应该懂技术，所以他们也许并不像你想象的那样难以说服，他们在被你说服后说不定比你还有热情。

把开源的思想灌输到高级技术人员的脑子里后，应该再次向上级提交一份经过改写的计划书。最好让那些赞同你的高级技术人员也签上名，或陪你一起去递交计划书，这样你不会显得势单力薄，更重要的是可以更有说服力。每打赢一场这样的战争都会削弱商业专利软件的霸权。

1.4 小结

本章回顾了开源软件的起源和MySQL崛起为一个世界级数据库管理系统的发展历程，讲解了什么是开源系统，并把它们与商业专利软件进行了对比。文中还介绍了开源许可证的来龙去脉，说明了作为全球开发人员社区里的一员应该承担哪些责任。

本章还对使用MySQL进行开发的问题做了简要的讨论，介绍了MySQL源代码的特点，给出了一些修改它们的指导原则。文中还讲到了MySQL AB公司采用双重许可证的做法以及对修改MySQL的行为有何约束。最后，讲述了一个把开源系统和商业产品集成在一起并取得巨大成功的例子。

下一章将以MySQL为例对关系数据库系统进行剖析，并介绍如何根据自己的需求对MySQL进行定制。本书的第二部分和第三部分将探讨MySQL的内部工作原理，以及它的源代码中最本质的部分。

想知道数据库系统内部发生着什么吗？你也许知道关系数据库系统（relational database system, RDBS）的基本概念，甚至是一位管理这类系统的专家，但你可能从未探索过数据库系统的内部工作原理。本书的读者中有很多人可能接受过培训并具备管理数据库系统的经验，但学校或职业培训课程在讲解数据库系统构造时往往一笔带过，并不深入讨论。数据库专业人员可能一辈子也不会用到这些知识，但知道系统的工作原理肯定是件好事，这些知识可以帮助你理解如何把服务器优化到最好，让你了解如何把它的功能发挥到极致。

本章将介绍RDBS的各个子系统以及如何构造这些子系统的基础知识。我将以MySQL为例对现代RDBS的核心组件进行剖析。如果你曾学习和研究过这些系统并希望赶快去看看MySQL的体系结构，请直接看2.4节。

2.1 数据库系统的体系结构

熟悉RDBS的内部工作原理，对于数据库的日常使用、维护和开发工作来说，并非必不可少的先决条件，但如果你想修改或扩展其功能的话，了解这类系统的组织方式就非常重要了。此外，数据库系统并非只有RDBS一种，如果你想知道其他类型的数据库系统与RDBS相比都有哪些优劣之处，掌握几种最流行的数据库系统的基本工作原理是很重要的。

2.2 数据库系统的类型

RDBS是目前最常见的数据库系统，但另外几种数据库系统也正变得越来越流行。本节将对三种最流行的数据库系统进行介绍和对比，它们是面向对象的数据库系统、对象关系数据库系统和关系数据库系统。只有了解了这些系统的体系结构和基本功能，你才会明白，MySQL AB公司把MySQL开发为一个开放源代码软件并向全世界公开其源代码的做法，是一件多么值得钦佩和赞赏的事情。如果不是这样，本书根本不可能向你展示这个魔盒里的宝物。

如果你熟悉这几种数据库系统，可以跳到2.3节。

2.2.1 面向对象数据库系统

面向对象数据库系统（object-oriented database system, OODBS）是支持面向对象编程（object-oriented programming）模型的数据存储和检索机制，它可以直接把数据当作对象来处理。这类系统包括真正的面向对象（object-oriented, OO）类型的系统，在应用程序和数据库之间传输的是不同类型

的对象。不过,大部分OODBS都没有标准的查询语言^①(一般通过编程接口来访问数据),所以还算不上是真正的数据库管理系统。

OODB是RDBS的有力竞争对手,尤其是在应用程序方面,RDBS的建模能力,或者说把数据以对象的形式存入表中的能力有所欠缺。有些应用程序包含大量的数据,并且不会将它们删除,因而就需要管理各个对象的历史。OODBS最独特的功能是它能够通过指定可经由某种OOP接口应用到复杂对象的相关操作和结构,来为复杂对象提供支持。

OODB特别适用于建立与现实世界尽可能一致的模型,不需要人们刻意地在各有关实体之间及其内部定义种种不自然的联系。面向对象的原理就是全面地并且通过建模的手段来再现真实的世界,这种再现对于处理那些变化迅速的事物(比如为某种会随时间而发生变化的东西建立模型)往往是必不可少的,尤其是在需要为结构化的数据添加OO功能的场合。不过,虽然宣称自己是一个OODBS的开源软件有很多,但其中有很大一部分不过是在关系数据库系统的基础上增加了几个查询语言接口的产物,与其说它们是OODBS,还不如说它们是带OO接口的关系数据库系统。严格地讲,真正的OODBS只允许通过某种编程接口去访问数据。

OO数据库系统的应用领域主要包括:地理信息系统(geographical information system, GIS)、科学和统计数据库、多媒体系统、图片存档和通信系统、XML仓库,等等。

OODB最受推崇的是数据(即对象)及其行为(即方法)是其专有的。大多数OODBS系统集成商都依赖OO方法来描述数据,并将这种表现形式应用于设计中来构建他们的解决方案。因此,OODB是为特定的实现而构建的,不会是通用的,也不像RDBS那样普遍具有“语句-响应-类型”接口。

2.2.2 对象关系数据库系统

对象关系数据库系统(object-relational database system, ORDBS)是OO技术与RDBS相结合的产物。ORDBS提供了一种机制,允许数据库的设计者为OO数据概念实现结构化存储和检索机制。ORDBS提供了关系模型的基础——含义、完整性、关系等,并扩展了这个模型,数据的存储和检索都围绕着对象来进行。ORDBS的具体实现有很多是纯概念性的,这是因为把OO概念映射到关系概念的理论 and 实践目前仍处于探索阶段。对关系数据库技术的修改或扩展,需要修改SQL语言使之支持对象类型、实体、封装操作和继承机制。然而,这些东西往往只是松散地映射到关系理论中的复合数据类型上而已。要知道,SQL语言再怎么扩展也无法提供真正的对象操作和OODBS的控制水平。目前最流行的ORDBS是ESRI公司推出的ArcGIS Geodatabase环境,其他例子还有Oracle和Informix。

ORDBS中的技术使用了基本的关系模型。绝大多数ORDBS是使用现有的商业关系数据库管理系统(如Microsoft SQL Server和Oracle)实现的。因为这些系统是建立在关系模型的基础上的,所以在把OO概念转换为关系机制的方面会遇到这样或那样的困难。下面是一些在使用关系数据库去开发面向对象应用程序时经常遇到的问题。

❑ 把OO概念模型映射到数据表不那么容易。

^① 有一些例外情况需要注意,但一般来说是这样的。

- ❑ 复杂的映射关系意味着复杂的程序和查询。
- ❑ 复杂的程序意味着维护问题。
- ❑ 复杂的程序意味着可靠性问题。
- ❑ 复杂的查询不容易优化，进而导致性能很低。
- ❑ 与关系系统相比，从对象概念到复合类型^①的映射更容易因为模式的变化而崩溃。
- ❑ `select all ... where`查询的OO性能比较低，因为它涉及更多的联接和查找操作。

这些问题看起来很麻烦，但解决起来并不困难：增加一个OO应用程序层作为关系数据库和OO应用程序之间的通信接口，让这个应用程序层去完成把对象转换为结构化（或永久性）数据存储的工作。有意思的是，这种做法违背了ORDBS的概念：这其实是在使用一种OO访问机制去访问数据，而创建ORDBS的目的是为了使用一种查询语言（及其扩展）在RDBS里存储和检索对象。

ORDBS与OODBS有很多相似之处，但这二者的内部原理有着很大的差异。OODBS是试图通过一种编程接口和平台把数据库功能添加到OO编程语言里去，ORDBS则是试图通过传统的查询语言及其扩展把更多的数据类型添加到RDBS里去。OODBS想要达到的目的是与OOP语言的无缝集成，ORDBS没有这么大的野心——它们通常需要一个中间应用程序层把来自OO应用程序的信息转发给ORDBS或底层的RDBS。还有，OODBS侧重于以OO技术为核心的应用程序，ORDBS侧重于对支持海量数据的基于对象的系统（如GIS应用）。最后，OODBS的查询机制以专用的OO查询语言操作对象为核心，ORDBS的查询机制更倾向于使用对SQL标准的扩展来快速检索大量数据。真正的OODBS都有经过优化的查询机制，如ODL（Object Description Language，对象描述语言）和OQL（Object Query Language，对象查询语言），而ORDBS使用的查询机制则是SQL查询语言的扩展。

ESRI公司的GIS应用程序套件产品里包含着一个名为Geodatabase（geographic database的缩写）的组件，该组件支持地理数据元素的存储和管理。Geodatabase是一个支持空间数据的ORDBS系统，是被实现为ORDBS的空间数据库的例子。

注解 ESRI公司把Geodatabase实现为ORDBS的做法并不意味着空间数据库系统只能用ORDBS或OODBS才能实现。GIS数据完全可以存储在一个支持空间数据的RDBS里！MySQL也可以用来存储空间数据，MySQL AB公司已经在它们的RDBS里增加了一个空间数据引擎。

ORDBS基于关系数据库平台是一个事实，但它们也提供了一些数据封装的层次和行为的层次。绝大多数ORDBS都是RDBS的特殊形式。提供ORDBS产品的数据库厂商通常通过修改SQL以包含对象描述符和空间查询机制的办法，来扩展“语句-响应”接口。这类系统基本上都是为了某种特殊的应用而建立的，所以它们的通用性也像OODBS那样受到了局限。

^① 尤其是在一个已经存储了不少信息的数据存储里改变对象的类型的时候。根据模式的变化情况，对象的行为有可能发生改变而不再有原来的意义。改变模式是一个需要深思熟虑的决策，但这个改动对ORDBS的影响，往往比对一般关系系统的影响严重。

2.2.3 关系数据库系统

RDBS是基于数据的关系模型（Relational Model of Data）的数据存储和检索服务，该模型是E. F. Codd于1970年提出的。这些系统是结构化数据的标准存储机制。正如C. J. Date在他的*The Database Relational Model: A Retrospective Review and Analysis*^①一书中所讨论的那样，人们已经进行了大量的研究去充实和细化Codd提出的原始模型。这个模型在理论和实践方面的发展历史在C. J. Date和H. Darwen合著的*Foundation for Future Database Systems: The Third Manifesto*^②一书中有最好的介绍。

关系模型是一个很直观的概念：把数据集中存储在某个地方（数据库），人们使用一种被称为查询语言的机制检索、修改和插入数据。因为有着全面系统的理论、坚实的数学基础和简单的结构，关系模型已经被很多厂商所实现。最常用的查询机制是SQL（Structured Query Language，结构化查询语言），它与自然语言很相似。虽然SQL没被收录在关系模型里，但在实际应用中，SQL却是RDBS里的关系模型不可缺少的组成部分之一。

在关系模型里，数据被表示为关于某个实体的一组相关信息（属性）。这些属性的值构成了一个数学意义上的集合，术语称之为元组（tuple，有时也称为记录）。元组被保存在包含具有相同属性集的元组的表里。这些表再通过域、键、属性和元组上的约束条件与其他表建立起关系。

元组与记录的区别

有许多人误认为记录是元组的俗称。二者有一个重要的区别：元组是一组元素的有序集合，而记录只是一组相关数据的无序集合。可是，列顺序在记录的概念里非常重要。有意思的是，在SQL语言里，查询结果可以是一条记录；而在关系理论里，每个结果都是一个元组。有许多书籍不加区别地互换使用这两个名词，许多人因此而形成上述错误的认识。

SQL是绝大多数RDBS首选的查询语言。SQL是在20世纪80年代作为一项标准而推出的，它目前已是一项行业标准。有不少人错误地以为SQL来自关系理论，因而是一个坚实的理论概念。这种错误认识的来源大概是因为几乎所有的RDBS都实现了SQL的某种形式。这种流行掩盖了SQL的许多不足，其中包括以下几点。

- ❑ SQL不支持关系模型所描述的域。
- ❑ 在SQL里，表可以有内容雷同的行。
- ❑ 查询结果（表）可以包含未命名的列和内容雷同的列。
- ❑ 有些数据库系统的SQL引擎对空值（缺失的数据）的处理有相互矛盾和不够完备的地方。有些人错误地把这一不足归咎于SQL，可实情却是SQL只负责把数据库提供的结果返回给用户^③。

在RDBS中使用的技术多种多样。有些系统偏重于优化关系模型的某个部分，还有些系统是用来优化数据模型的某些应用。有些RDBS只能完成简单的数据存储和检索工作，而高级的RDBS应用程序套件能够对复杂的数据、过程和工作流进行处理。你可以用一个简单的数据库来保存家中光碟

① Addison-Wesley公司2001年出版。

② Addison-Wesley公司2000年出版。

③ 有些数据库系统对空值的处理办法很不直观，而有些甚至可以用荒诞这个词来评价。

的相关信息，可以用一个数据库来管理宾馆的客房预订系统，也可以用一个复杂的分布式系统来管理网上的信息。正如我在第1章里提到的那样，正是由于MySQL成为存储有关数据的数据库，许多Web应用（尤其是采用了Web 2.0技术的）才得以在LAMP组合上实现。

Web 2.0

Web现在已经发展到了能够让人们共享信息和在线合作的程度，人们创造了Web 2.0这个热门词来描述这一巨大变化。扩展了全球电子社区的概念的应用也因此被人们称为“Web 2.0应用”。这方面的例子包括图片共享、博客以及信息和视听服务，等等。这类应用与近十年的Web技术进步息息相关，而LAMP就是其中之一。有许多Web 2.0应用是采用开源解决方案实现的。Web 2.0技术仍在发展和完善当中，但它对因特网的前景必将产生深远的影响。

关系数据库系统提供了最健壮的数据独立性和数据抽象性。通过使用“关系”这个概念，RDBS提供了一种真正通用的数据存储和检索机制。与此相应的是，这些系统也变得极其复杂，需要专家来建立和维护。

下一节将对RDBS的体系结构及其各组成部分进行剖析。之后，还将剖析一个RDBS的具体实现（MySQL）。

MySQL是一个关系数据库系统吗

许多数据库理论家会说真正的RDBS非常之少。他们还会指出是不是关系数据库，在很大程度上要取决于你对数据库系统所支持的功能的定义，而不是那个系统与Codd的关系模型有多么相符。

从宣传广告上看，RDBS应该具备的关键功能MySQL几乎都能提供。这包括可以使用外键把表与另一个表关联起来，能够提供一个关系化的数据查询机制，具备索引和缓存功能，等等。显然，MySQL具备的功能不限于这些。

那么，MySQL到底是不是一个RDBS呢？这要取决于你对“关系”的定义。从MySQL提供的功能上看，它是一个RDBS；如果严格按照Codd的关系模型来分析，MySQL还缺少这个模型所要求的一些功能。可话又说回来了，许多其他的RDBS也是这样的。

2.3 关系数据库系统的体系结构

RDBS是一种由特殊机制组成的复杂系统，这些复杂机制专门用来处理存储和检索信息所需的全部函数。人们经常把RDBS的体系结构与操作系统的体系结构相提并论。如果观察一下RDBS（尤其是在那些拥有大量客户端的服务器上）的工作情况，就会发现它们与操作系统有许多共同点。比如说，有多个客户端意味着这个系统必须处理许多请求，而这些请求有可能会在同一时间访问被存储在同一个位置（比如同一个表）的数据。因此，RDBS必须具备高效的并发处理能力。类似地，RDBS必须迅速地向每一个客户端提供它们所检索的数据，这通常是利用文件缓存技术，将最新或最常用的数据保存在内存里而实现的。进行并发处理所需要的内存管理技术与操作系统中的虚拟内存子系统很相像。RDBS与操作系统的其他类似之处还包括网络通信支持，以及为最大限度地提升

数据查询命令的执行性能而专门设计的优化算法。

我将从用户的角度，从检索数据的查询操作开始对RDBS体系结构的探索之旅。在接下来的几节里，你可以跳过已经熟悉的内容去阅读自己感兴趣的东西。但我建议大家把所有小节都读一遍，因为它们将向你全面展示一个典型的RDBS的构造细节。

2.3.1 客户端应用程序

绝大多数RDBS客户端应用程序都是与服务器分离的可执行程序，它们通过一条通信路径（例如套接字或管道等某种网络协议）与数据库连接。另外一些通过编程接口直接连接到服务器系统，此时的数据库系统变成了客户端应用程序的一部分，这类数据库被称为嵌入式系统。对嵌入式数据库系统的详细讨论见第6章。

在那些通过一条通信路径去连接数据库的系统当中，绝大多数是通过一组被称为数据库连接器的协议建立连接的。数据库接口大都基于ODBC（Open Database Connectivity，开放数据库连接）模型^①。MySQL还支持用于Java（JDBC）和Microsoft .NET环境的连接器。多数ODBC连接器也支持网络协议上的通信。

什么是ODBC

ODBC是一种标准化的应用程序编程接口（application programming interface, API）。ODBC的基本用途是把SQL命令传输到数据库服务器，再把检索到的信息返回给发出SQL命令的应用程序。ODBC的具体实现包括一个使用了用来和ODBC交互的API的应用程序、一个支持API的核心ODBC库，以及一个针对特定数据库系统的数据库驱动程序。我们通常把上述三个组件统称为一个“ODBC连接器”。ODBC接口相当于客户端应用程序和数据库服务器之间的一个中转站。ODBC已经成为几乎所有关系数据库系统（以及绝大多数面向对象的关系数据库系统）的标准组件。人们已为各种客户端和数据库系统开发出了数百种连接器和驱动程序。

在提到客户端应用程序，我们所指的通常是那些用来向服务器发送和从数据库服务器接收数据的程序。其实，我们用来配置和维护数据库服务器的应用程序也属于客户端应用程序的范畴。这些工具中的绝大多数与其他数据库应用程序一样，也需要通过一条网络路径去连接数据库服务器，它们大部分使用ODBC连接器或是诸如JDBC（Java Database Connectivity，Java数据库连接）之类的变体，少数使用专用的协议来管理数据库服务器和完成特定的管理/维护工作，还有一些（例如phpMyAdmin）使用一个端口或套接字。

不管它们的具体实现是怎样的，客户端应用程序的基本用途是一样的：向数据库系统发出命令并获取那些命令的结果，解释和处理那些结果并把它们呈现给用户。标准的命令语言是SQL。客户端通过ODBC连接器向服务器发出SQL命令，ODBC连接器使用由驱动程序指定的网络协议把命令传输给数据库服务器。这个过程如图2-1所示。

^① 有些人把“ODBC”解释为“Object Database Connectivity”或“Online Database Connectivity”，但最为人们所接受的定义还是“Open Database Connectivity”。

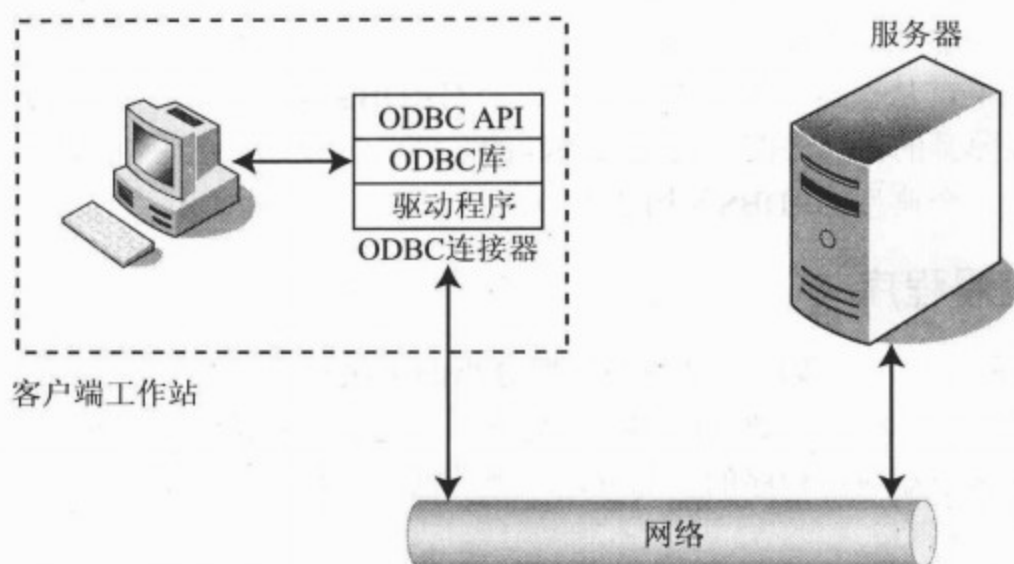


图2-1 客户端应用程序/数据库服务器之间的通信

2.3.2 查询接口

SQL等查询语言是一种用来向数据库系统“提问”的语言（有自己的语法和语义）。事实上，对SQL的使用被认为是数据库系统取得成功的主要原因之一。SQL可以细分为几种子语言，它们共同构成了数据库系统应用的坚实基础。数据定义语言（data definition language, DDL）是数据库专业人员用来创建和管理数据库的，主要任务包括创建和修改各种表、定义各种索引、管理各种约束条件。数据操纵语言（data manipulation language, DML）被数据库专业人员用来查询和更新数据库里的数据，主要任务包括添加、更新和查询各种数据。DDL和DML命令加在一起占了数据库系统所支持的命令中的大多数。

SQL命令有各自的专用语法。下面是SQL语言中的SELECT命令的语法，其中的斜体字是需要用户输入的变量，方括号（[]）里的内容是可选的参数。

```
SELECT [DISTINCT] listofcolumns
FROM listoftables
[WHERE expression (predicates in CNF)]
[GROUP BY listofcolumns]
[HAVING expression]
[ORDER BY listof columns];
```

这条命令的语义如下^①。

- (1) 为FROM子句里的表生成笛卡尔积（集合乘法），准备根据其他子句里的条件从中筛选出查询结果。
- (2) 如果WHERE子句存在，按该子句里所有的表达式到FROM子句列出的表里筛选数据。
- (3) 如果GROUP BY子句存在，按该子句给出的属性对查询结果进行分组。
- (4) 如果HAVING子句存在，按该子句给出的条件对分组后的查询结果进行过滤。
- (5) 如果ORDER BY子句存在，按该子句给出的条件对查询结果进行排序。

^① M. Stonebraker和J. L. Hellerstein, *Readings in Database Systems*, 3rd ed., Michael Stonebraker编著(Morgan Kaufmann Publishers, 1998)。

(6) 如果DISTINCT关键字存在，从查询结果里剔除内容雷同的行。

上面这段代码示例是绝大多数SQL命令的代表；所有这些命令都有必须给出的部分，其中绝大多数还有可选的部分和基于关键字的修饰符。

在查询语句通过网络协议被传输到服务器[称为运送(ship)]之后，数据库服务器就将开始解释和执行有关的命令。从此时开始，查询语句将被直接称为查询，因为它代表着一个要求数据库系统提供答案的问题。不仅如此，在接下来的几个小节，我将假设“查询”是一条SELECT命令，它代表用户发出的一个数据请求。所有的查询不管它是数据操作还是数据定义，都将沿着同样的路径通过系统。从现在开始，我们将讨论在数据库服务器的内部发生的事情，而整个流程的第一步是分析客户端在请求些什么——对查询进行分析，并把它分解为一系列最基本的可执行元素。

2.3.3 查询处理

在基于客户端/服务器模型的数据库系统的环境下，数据库服务器负责处理来自客户的查询并把查询结果返回给客户。有人把这一过程形象地比喻成查询换结果，先把查询“运”到服务器，再把换来的结果(数据)“运”回客户端。这么做的直接好处有两个：其一是可以减少查询的通信时间；其二是可以利用比客户端丰富得多的服务器资源来完成查询。此外，这个模型还把数据在服务器上的存储和检索方式，与数据在客户端的具体使用情况分成两个互不影响的部分。换句话说，客户端/服务器模型支持数据独立性。

数据独立性(data independence)是Codd于1970年提出的关系模型的主要优点之一：把物理实现与逻辑模型分开。Codd是这样说的^①：

大型数据库^②必须做到让它的用户不必去了解数据在机器里是如何存储的……当数据的内部表示发生变化时，最终用户的操作活动和绝大多数应用程序不应该受到任何影响。

这种分离使得程序员可以专注于逻辑模型的开发而无须考虑它们的物理实现。数据独立性(被Elmasri和Navathe称之为“数据的物理独立性”^③)的目标是让每一个逻辑元素都独立于所有的物理元素(参见表2-1)。比如说，数据的逻辑层——关系(表)以及按元组(行)排列的属性(字段)，与数据在存储介质上是如何存储的完全无关。

表2-1 数据库设计方案的逻辑模型和物理模型

逻辑模型	物理模型
查询语言(query language)	排序算法
关系代数(relational algebra)	存储机制
关系演算(relational calculus)	索引机制
表达式和变量(relvars)	数据的表示形式

数据独立性带来的挑战之一是数据库的编程工作需要完成两个过程：其一是写出逻辑查询(logical

① C. J. Date, *The Database Relational Model: A Retrospective Review and Analysis* (Reading, MA: Addison-Wesley, 2001)。

② 原文此处为data bank，是database的同义词。——编者注

③ R. Elmasri和S. B. Navathe, *Fundamentals of Database Systems*, 4th ed. (Boston: Addison-Wesley, 2003)。

query), 把查询要干什么描述清楚; 其二是写出物理计划 (physical plan), 把逻辑查询的实现步骤描述清楚。

逻辑查询通常可以写成许多不同的形式, 你可以用SQL之类的高级语言来写, 也可以用代数查询树^①来表示。比如说, 在传统的关系模型里, 逻辑查询可以用关系演算和关系代数来描述。关系演算比较适合用来描述需要计算什么。关系代数则是提供了一个算法 (但这个算法没有涉及许多与查询处理有关的细节) 来让你找到想要查询的东西。

物理计划是一棵以数据库系统的查询执行引擎能够理解和处理的方式实现出来的查询树。查询树是一种树状结构, 它的每个结点包含着一个查询操作符并有一系列子结点, 子结点的个数与查询操作所涉及的表的个数相对应。这种查询树可以由数据库系统的优化器转换为一个执行计划, 我们可以把这个计划想象成一个可以用查询执行引擎去执行的程序。

查询语句在经过语法分析、错误检查、优化和计划的生成/编译这几个阶段后才会执行。图2-2给出了一个典型的数据库系统所使用的查询处理步骤。语法分析步骤负责检查查询语句的语法是否正确并确定它是哪一种查询操作。解析器 (parser) 将把查询输出为某种内部格式以方便优化器生成一个高效的查询执行计划。执行引擎负责具体执行各个查询并把结果返回给客户端。整个过程如图2-2所示: 分析工作完成后, 对查询进行错误检查, 然后进行优化, 再挑选一个计划并编译, 最后执行查询。

这个过程的第一步是把逻辑查询从SQL语句转换成关系代数里的一棵查询树。这一步由解析器负责完成, 常见的做法是先把SQL语句拆分成若干部分, 然后构建成一棵查询树。下一步是把逻辑代数中的查询树转换成一个物理计划。一般来说, 同一棵查询树可以对应于许多种等价的物理计划。

寻找最佳执行计划的过程被称为查询优化, 即根据查询的某些执行性能指标 (例如执行时间) 把有着最佳执行性能的计划找出来。查询优化步骤的目的是从优化器的搜索空间内找出一个最佳或接近最佳的计划来。优化器将先把关系代数中的查询树复制到它的搜索空间里, 再通过生成新的执行计划来扩展其搜索空间 (扩展次数有一个上限), 并从中找出最佳的计划 (执行速度最快者)。

从此种意义上讲, 我们可以把优化器视为SQL查询命令编译器的代码生成部分。事实上, 有些数据库系统的编译步骤的确会把查询转换成为一段可执行程序, 但绝大多数数据库系统是把查询转换成某种可以使用内部执行步骤库来执行的格式。此时, 代码编译步骤生成的代码将由查询执行引擎解释。与传统意义上的编译器相比, 优化器的侧重点是生成“非常高效”的代码。比如说, 优化器将使用数据库系统的目录去收集与本次查询所涉及的关系有关的信息 (例如元组的个数), 而传统的编程语言编译器一般不会这么做。最后, 优化器把它找到的最佳物理计划从它的内存结构里复制出来并发送给查询执行引擎。查询执行引擎将以存储在数据库里的关系作为输入来执行这个计划, 与查询条件相匹配的各行所组成的表就这样被检索出来了。



图2-2 查询的处理流程

① A. B. Tucker, *Computer Science Handbook*, 2nd ed. (Boca Raton, FL: CRC Press, 2004)。

所有这些活动都要花费时间和消耗资源，所以数据库的实现者必须把查询优化器和执行引擎的性能作为整体效率中的一个因素来考虑。因为以不同的访问方法（读取数据的方法）和不同的执行顺序组合出来的候选执行计划的数量很大，所以这种优化的开销非常巨大。因此单个查询也有可能产生接近无限个执行计划。但是，数据库系统通常都会把这个问题与一些众所周知的最佳实践放在一起。

生成大量查询计划的主要原因之一是，优化工作是在某些重要的性能指标参数未知或无法确定的情况下进行的。为了保证优化工作的进行，数据库系统通常会在以下（但不仅限于此）几个方面作出一些特定的假设：数据库的内容（例如关系属性中的值分布情况）、物理模式（例如索引的类型）、系统参数的值（例如可用缓冲区的个数）和查询常数的值等。

2.3.4 查询优化器

有些人错误地认为查询执行阶段的所有步骤都是由查询优化器完成的。但你即将看到，查询优化只不过是查询执行阶段的步骤之一。以下内容将对查询优化器进行详细的描述并揭示它在查询执行过程中所扮演的角色。

查询优化是查询编译过程的一个环节，它负责把以某种高级的非过程化语言（如SQL）写出的数据操纵语句，转换为一个更详尽的过程化的操作符序列，这个序列就是所谓的查询计划。查询优化器的任务是对各候选计划的开销进行估算，并从中选出一个开销最小的（执行最快者）。

按“基于计划”的方式进行查询优化的数据库系统往往假设，许多计划都可用来产生任意给定的查询。事实也的确如此，但按照这些计划去执行查询时，可能消耗的资源 and 需要花费的时间就未必相同了。既然如此，就需要把开销最小和/或用时最少的那个计划找出来。在设计一个嵌入式集成系统或一个需要在小型平台（资源较少）上提供高流量或高速数据服务的系统时，怎样才能使资源占用量和执行开销之间取得平衡是经常会遇到的一个问题。

基于计划的查询处理策略如图2-3所示，查询将沿着图中的箭头方向移动。SQL命令先被传递到查询解析器接受分析和错误检查，并被转换为一种内部表示；内部表示通常是一个关系代数表达式或一棵查询树（参见前面的讨论）。接下来，查询被传递到查询优化器，优化器将对所有等价的关系代数表达式进行检查并为每一种组合分别生成一个计划。优化器将选出开销最小的那个计划并把查询传递到代码生成器，后者将把查询翻译成一种可执行的格式——也许是可直接执行的代码，也许是某种以解释方式执行的中间代码。最后，查询处理器执行查询并逐条返回结果集（result set）里的行。

这是一种常见的实现策略，在绝大多数数据库系统上都能看到这一流程。随着计算机硬件技术的进步，不同的查询计划在预计开销方面差距巨大的情况已非常少见。事实上，绝大多数查询计划的预计开销都相差不大。这种情况使得一部分数据库实现者开始采用另外一种策略来实现查询优化器：在查询优化步骤使用一些大家公认的好规则（称为启发式）和好经验去进行查询优化，这被称为“启发式优化策略”。有些数据库系统混合使用多种优化技术，即以某一种技术为主并吸收其他技术的长处。

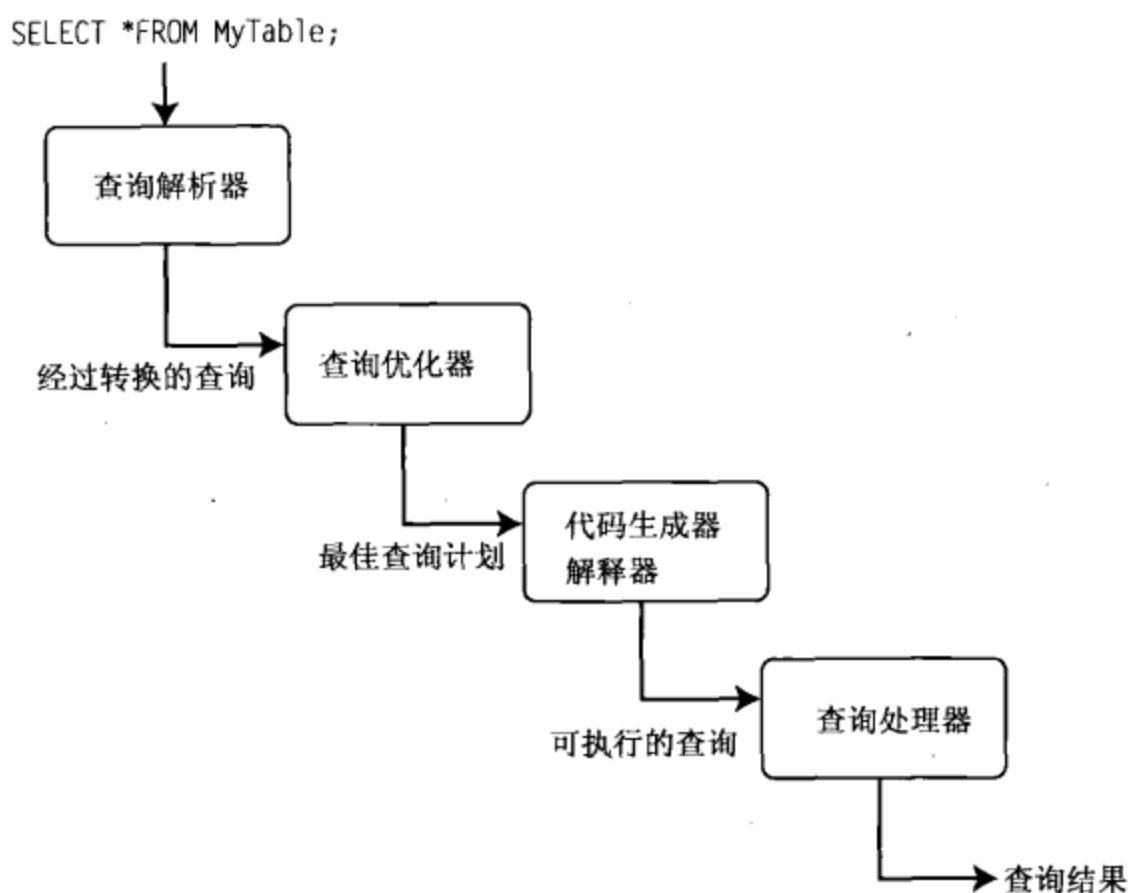


图2-3 基于计划的查询处理流程

进行查询优化的4种基本技术是：

- ☐ 基于开销的优化；
- ☐ 启发式优化；
- ☐ 语义优化；
- ☐ 参数优化。

没有一种优化技术可以保证优化出最佳的执行计划，所有这些方法的目标是在保证正确结果的前提下让查询能够高效率地得到执行。

基于开销的优化器先根据一系列等价规则为给定查询生成一系列查询评估计划，然后再根据关于执行这个查询所需要的关系和操作的性能指标数据（或统计数据）中选出开销最小的那一个。一般来说，查询越复杂，可供选择的等价计划就越多。基于开销的优化技术的目标，是最大限度地利用索引和从过去的查询中收集到的统计数据去安排查询的执行和表的访问。Microsoft SQL Server和Oracle等系统使用的就是基于开销的优化器。

启发式优化器先根据一系列规则把查询改写成最优的形式，然后才开始生成和挑选执行计划。运用那些规则的目的是为了消除那些看起来不太高效的查询，从而保证在此基础上生成的查询计划在接评估之前就已经是（但不绝对）比较优秀的了。启发式优化技术的目标是用规则来保证“好”的执行效果。使用启发式优化器的系统包括Ingres和各种学术研究性的变体。这类系统往往把启发式优化技术当作一种避免糟糕计划的手段，而非一个主要的优化手段。

语义优化技术的目的是：根据一条查询命令所包含的语义（或拓扑结构）和有关数据库/关系/索引的实际情况而生成的查询执行计划，可以保证这条查询命令在给定数据库上以最优的方式得到执

行。虽说目前还没有使用语义优化技术作为其主要优化手段的商业数据库系统,但这种技术正日益成为人们研究的焦点。作为语义优化技术的前提,优化器需要对数据库的构造和当前情况有一个基本的了解。在收到一条查询命令时,优化器将根据自己对系统约束的了解,在确定查询会返回一个空结果集后简化或忽略它。人们对这种技术寄予了很大希望,认为它在未来的RDBS里能够让查询的处理效率得到更大的提高。

参数优化技术是启发式方法和基于开销的优化技术相结合的产物,它提供了一种方法来生成可预估开销的有效查询计划的集合,因此能从中选出一个开销最小的计划去执行。

MySQL就是一个混合使用多种优化技术的数据库系统。MySQL的查询优化器是围绕“选取-投影-联结”策略而设计出来的,它综合了基于开销的优化器和启发式优化器的优点,生成的查询计划相对较少,然后再利用基于开销的优化技术从中选出一个最短的执行路径。这一策略可以从整体上保证一个“好”的执行计划,但不能保证生成最好的计划。实践证明,这一策略对运行在不同环境里的许多种查询来说效果都很好。实践还证明,MySQL的内部表示完全可以满足最大型的生产数据库系统在速度方面的要求。

微软的SQL Server是一个使用基于开销的优化器的数据库系统。SQL Server的查询优化器是围绕着一个传统的基于开销的优化器设计出来的,它将把查询语句翻译为一个可以高效执行并返回相关结果的过程。该优化器根据从以前执行过的查询收集到的信息或统计数据以及数据库里的数据特征,为同一个查询生成多个等效的过程,再利用统计数据去预测哪一个过程的执行效率最高^①。一旦确定了最有效率的过程,就开始执行并把结果返回给客户端。

不确定参数(比如用户输入)会让查询优化工作变得复杂。比如说,如果某个存储过程(stored procedure)里有一条查询命令,该命令在这个存储过程执行时接受一个来自用户的参数,那么这个参数就是一个不确定参数(unbound parameter)。在这种情况下,对查询进行优化几乎是不可能的;如果无法在执行前收集到一些关于输入的信息,优化器将很难生成一个开销最小的执行计划。反过来说,哪怕一点点信息都会对优化工作有帮助。如果只有很少几条记录满足输入,甚至一个基本的索引也远优于文件扫描。如果满足输入的记录很多,使用索引的效果就更好了。如果输入不确定,优化器将无法在优化阶段作出选择——挑选最优执行计划的工作将推迟到执行开始以后。

可以这样解决选择问题:把输入当作一个开放变量,让优化器根据以前的查询执行情况以及来自基于开销的优化器的统计数据,把有可能发生的执行计划全部生成出来。对解决这个问题有帮助的统计数据包括该不确定参数的属性的频率分布情况。

2.3.5 查询的内部表示

在数据库系统的内部,查询可以被表示为其原始SQL命令的几种替代形式。存在这些替代形式的原因包括:原始SQL命令的冗余、子查询和联结操作在特定约束条件下的等价性以及WHERE子句里的表达式之间的逻辑运算结果等。查询有多种替代形式给数据库实现者带来了一个问题:不管用户写出的

^① 利用统计数据进行优化的做法起源于最初的基于开销的优化器。事实上,商业数据库里的许多工具就是为了分析和生成这些统计数据而存在的,数据库专家可以利用这类数据去优化数据库的结构,让查询优化工作更有效率。

查询命令是什么样的，查询优化器都必须为它选出一个最优的执行计划。

一旦查询优化器构造出了一个高效的执行计划（启发式或混合式优化器）或选出了一个最有效率的计划（基于开销的优化器），查询就将被传递到下一个处理环节——执行。

2.3.6 查询的执行

数据库系统可以使用好几种方法来执行一个查询。大部分数据库系统使用迭代或解释执行策略。

迭代方法提供了一些途径来生成一个可直接执行的函数调用序列来完成各种基本处理操作（联结、投影等），并没有包含内部表示的特性。用来把查询转换为迭代方法的技术主要有函数编程和程序变换。把基于关系代数的查询条件转换成迭代程序的算法有好几种。比如说，有些算法先把查询条件转换为递归程序，再利用一系列变换规则对它进行简化，然后生成一个执行计划。还有一种算法使用两个步骤来完成这一工作：第一步使用一个较小的变换规则集对查询的内部表示进行简化，第二步使用程序变换技术生成等效的函数，然后生成执行计划。

上述机制的具体实现办法是：先用某种高级语言编写出一系列最基本的功能函数，再用一个调用栈或过程调用序列把它们链接在一起。在找出最佳的查询执行计划之后，用一个编译器（通常就是用来创建该数据库系统的那个）把这些过程调用编译成一段可执行的二进制代码。因为迭代方法的开销比较大，所以编译好的执行计划通常会被存起来供类似或相同的查询重复使用。

另外，解释方法使用现有的已编译基本操作抽象来生成查询执行计划。在选出最佳查询执行计划后，先把它重新构造成一个方法调用队列，然后从这个队列里依次取出各个方法进行处理。各个方法的执行结果放在内存里供后续方法做进一步处理。这种策略的实现经常称为懒汉策略^①，这是因为虽然队列里的每一个方法都经过了编译和优化，但那些优化并不针对某个特定的查询，所以它们组合在一起的性能就不一定是最优的。采用解释方法来执行查询的数据库系统占绝大多数。

在这里，编译这个概念很容易引起混乱。有些数据库专家认为，对查询进行已编译的查询就是把迭代查询执行计划实际编译成一段可执行代码；但在Date的著作里，对查询进行编译只是对它进行优化并保存起来供今后执行而已。因为MySQL的查询优化器和执行引擎不会保存查询执行计划供今后重复使用（MySQL的查询缓存机制是个例外），而且查询执行引擎实际执行的也不是任何一种编译或汇编代码，所以我在后面的讨论里将尽量避免使用编译这个词。有意思的是，MySQL中的存储过程与Date对“编译”概念的解释不谋而合：对存储过程进行编译（或优化）并把它们保存起来供今后使用，它们可以在满足其输入参数的数据上运行多次。

在查询执行阶段，查询执行引擎将对查询树（或由内部结构表示的查询）的每一个部分进行处理并为每一个部分调用有关的方法。这些方法对应着关系代数里定义的各种操作，如投影、限制、并、交，等等。对于这些操作中的每一个操作，查询执行引擎都将调用一个相应的方法对输入数据进行处理，然后把数据传递到下一步。比如说，投影操作只返回有关数据的某几个属性（即列）——查询执行引擎将把属性不符合筛选条件的数据排除掉，只把剩余的数据传递到查询树（即结构）的下一个操作。表2-2列出了几种最常见的操作并对它们做了简要的解释。

^① lazy evaluation，也被称为“懒惰计算法”或“惰性求值”。——编者注

表2-2 查询操作

操 作	说 明
限制	返回与WHERE子句里的条件（断言）相匹配的元组（有些系统对HAVING子句的处理也与此相同或类似）。这个操作通常被定义为SELECT（选取）
投影	只返回有关元组的列清单里列出的属性
联结	返回与联结条件（也叫“联结断言”）相匹配的元组。联结操作的种类有很多，详见下面的文本框内容

2

联结操作

联结（join）操作有好几种类型，很容易混淆，有些数据库专家会千方百计地避免使用它们。SQL语言有很强的表达能力，很多联结操作都可以简单地写成WHERE子句里的表达式。虽说绝大多数数据库系统都可以正确地把这些查询转换为联结操作，但这多少有点儿“偷懒”的味道。下面列出了你在RDBS里有可能会遇到的各种联结操作的类型。联结操作可以有联结条件（条件联结）、要求两个属性值必须相等（同等联结）或无须满足任何条件（笛卡儿积）。联结操作可以划分为以下几大类别。

- 内联结：对两个关系进行联结，返回匹配的元组。
- 外联结（左、右、全），至少返回FROM子句所列出的其中的一个表或视图的全部行——只要那些行满足全部的WHERE搜索条件。左外联结操作将返回位于联结操作符左侧的那个表的全部行；右外联结操作将返回位于联结操作符右侧的那个表的全部行；全外联结操作将返回位于联结操作符左、右两侧的两个表的全部行。不相匹配的行的属性值将返回为空值。
- 右外联结，对两个关系进行联结，返回匹配的元组和位于联结操作符右侧的那个表的全部元组，来自另一个关系的未匹配属性将全部返回为空值（null）。
- 全外联结，对两个关系进行联结，返回两个关系里的所有元组，来自另一个关系的未匹配属性将全部返回为空值（null）。
- 叉积，对两个关系进行联结，把第一个关系里的每一个元组映射到另一个关系里的全部元组上。
- 并，对两个有着同样结构的关系进行联结，只返回满足条件的匹配，相当于数学集合理论中的“并”操作。
- 交，对两个有着同样结构的关系进行联结，只返回不满足条件的匹配，相当于数学集合理论中的“交”操作。

确定了如何执行一个查询（或选中的查询计划）只是问题的一半，还需要考虑的其他事情是如何访问那些数据本身。从磁盘（文件）读写数据的办法有很多，但选择哪个办法取决于查询的目的。文件访问机制的作用是降低从磁盘访问数据的开销，提高查询执行的性能。

2.3.7 文件访问

文件访问机制（file-access mechanism）也叫数据库的物理设计（physical database design），它从数据库系统开发的早期时代就非常重要。不过，因为操作系统提供的通用文件子系统的效率和简单性，

文件访问机制的重要性已经降低了。现在，文件访问只是文件存储和索引创建方面的最佳经验的运用而已，比如把索引文件和数据文件分开并放在不同的磁盘输入/输出（I/O）系统上以改善性能等。为了让数据库能够根据特定应用程序的需求进行定制，有些数据库系统使用了不同的文件组织技术。MySQL在这方面大概是独一无二的，它支持的文件访问机制（称为存储引擎）非常多。

目标很明确：必须最大限度地降低数据库系统的I/O开销。这包括选用适当的磁盘数据结构以通过高效率的访问路径又快又准地检索出有关的数据，对磁盘上的数据进行组织以最大限度地降低检索有关数据的I/O开销等。总体性能目标是减少磁盘访问（或磁盘I/O）操作的次数。

关于如何设计一个数据库系统的技术有不少，但专门针对文件访问机制（数据文件的实际物理实现）的技术并不多。不仅如此，有许多研究人员认为，最优化的数据库设计（从物理的角度看）不可能真的实现，也不应该过份强求。他们的观点是：现代磁盘子系统的性能已得到极大的提高，这方面的技术和研究已足以让数据库的实现者们满足潜在用户的需求。

如果你想创建一个性能优良的结构，就必须考虑许多因素。早期的研究者们认为应该根据数据的内容或上下文把数据划分为一系列子集合。比如说，在一个人事档案数据库里，应该把部门编号相同的所有数据集中划分为一组，把这些数据和它们对相关数据的引用（指针）保存到同一个地方。按照这一思路，集合可以组织在一起构成超集，最终在文件内部和文件之间形成一种层次化的结构。

在这种层次化结构里访问数据的过程是：从位于最高层次的集合开始扫描，只扫描为获得必要的数而必须扫描的集合。这种做法大大地减少了需要扫描的元素个数。把同类数据集中保存在一起可以最大限度地缩短搜索时间。磁盘上的数据在结构化文件里的布局叫作文件组织（file organization）。文件访问机制的设计目标是找到这样一种访问方法：能连续、快速地完成一连串数据处理操作，使数据库里的数据随时都能反映出现实世界的情况。

为了保证更高效的存储和检索，文件组织技术随着操作系统的发展也在进步，但许多目前被普遍接受的方法在现代数据库系统中已显得有些力不从心，这在那些配有高速大容量硬盘和高吞吐量的系统上体现得尤其明显。对数据库设计方法的理解和进一步研究，无论是来自书本还是来自实践，都可以帮助我们开发出更好的数据库系统。比如说，随着采用冗余和分布式技术的系统日益普及，新硬件和/或数据可用性、安全性和恢复技术等领域里的研究工作正变得越来越热门。

从磁盘访问数据的开销很大，而使用某种缓存机制（有时称作缓冲区）可以大大提高从磁盘读取数据的效率，降低数据的存储和检索开销。比较常见的做法是通过预测下次磁盘读操作或者根据某个算法把数据提前拷贝到缓冲区里，目的是让最频繁使用的数据驻留在内存里以加快数据访问速度。能否高效处理好磁盘和主存的差异是评判数据库系统质量高低的重要标准之一。使用磁盘或使用主存对数据库性能的影响不可不知。表2-3对主存与磁盘的性能表现进行了汇总。

表2-3 性能表现

问 题	主存与磁盘
速度	主存至少比磁盘快1000倍
存储空间	在同样的成本下，磁盘的容量要比内存大数百倍
持久性	电源被关闭后，磁盘上的数据不会丢失，主存里的数据全都没有了
访问时间	主存只需几纳秒就可以开始发送数据，磁盘则要经过几毫秒
数据块的大小	主存可以一次存取一个字，磁盘则是每次读写一个数据块

数据库物理存储技术的进步对任何一种存储策略和缓存机制都有所改善，但物理存储技术基本要素的探索性研究工作的进展并不大。有些人在从硬件入手去研究如何利用新硬件，有些人在从实用角度出发去研究人们到底想存储些什么。持久性问题已很少有人研究，这或许是因为操作系统在这方面已经相当完善和高效的缘故吧。

文件访问机制用来存储和检索数据库里保存的数据。绝大多数文件访问机制都有额外的功能层来快速地确定数据在文件里的位置。这些层次被称为索引机制。索引机制将根据数据的某个组成部分（键）来确定有关数据的存放位置并提供相应的访问路径（搜索和检索数据的办法）。简单的索引机制可以是一个简单的键列表，复杂的索引机制会有一些用来加快速度的复杂结构。

我们的目标是又快有准地找到需要的数据，尽量减少从磁盘读取数据块的个数。这可以通过把标识数据的值（或键）和记录在磁盘上的存放位置保存起来，从而为有关数据生成一个索引的办法来实现。显然，读取索引数据要比读取全部数据快很多。使用索引的最大好处是可以高效率地搜索大量的数据，用不着把每个数据项都读取一遍就能找到想要查找的东西。索引机制会提供一系列用来对磁盘上的数据文件进行搜索的方法。这些方法有的可以加快数据的随机访问速度，有的可以加快数据的顺序访问速度。

索引机制有许多种类。它们大都采用某种树结构来存储键和磁盘块地址。例子包括B-树、B+树和散列树。与这些结构配套的是一些能够在最短的时间里从这些结构里找到特定键的遍历算法。绝大多数数据库系统在它们的索引机制里使用的是B-树的某种变体。这些树算法的搜索速度非常快，占用的内存空间却不大。

在查询执行阶段，解释型查询执行方法将通过索引机制调用某个特定的访问方法去请求数据。随后，执行方法读出数据（通常是每次一条记录）、根据对表达式的计算来判断查询是否与断言相匹配、接着对数据进行必要的处理、最后把数据传递到服务器的传输部分并由后者把数据发送回客户端。

2.3.8 查询结果

把与本次查询有关的元组全部检索出来并对它们进行了必要的处理后，那些元组将沿着同样的通信路径（有时会是另外一条）返回给客户端。那些元组随后被传递到ODBC接口进行封装并呈现在客户端应用程序里。

2.3.9 关系数据库的体系结构小结

本节介绍了通过一个典型的关系数据库系统体系结构查询数据的一系列步骤。正如你所看到的，查询从客户端发出的一条SQL命令开始，经ODBC接口通过一条通信路径（网络）被传递到数据库系统。在查询被分析，转换为一种内部结构，优化和执行之后，查询结果将被返回给客户端。

既然已经介绍了查询处理的所有步骤，大家也看到了数据库系统各子组件的复杂性，现在该看一个实际的例子了。接下来的2.4节将剖析MySQL数据库系统的体系结构。

2.4 MySQL 数据库系统

MySQL的源代码组织得非常好，也使用了很多结构化的类（有些是复杂的数据结构，有些是对象，但大部分是结构），但整个系统并不是一个真正的模块化体系结构。记住这一点在分析MySQL的体系结构时很重要，在研究MySQL的源代码时更重要。这句话的意思是说，你在MySQL的源代码里有时

会发现体系结构元素之间没有明确的界线。关于MySQL源代码的更多信息（包括如何获得它）请参见第3章。

虽然有些人把MySQL的体系结构描述为一个基于组件的系统，是由一组模块化的子组件构成的，但实际上它既不是基于组件的，也不是模块化的。虽然MySQL的源代码是用C和C++语言混合编写的，这个系统里的许多功能也是用对象实现的，但从严格的面向对象编程意义上讲，这个系统并不真正地面向对象。这个系统建立在一系列函数库和数据结构的基础上，这些函数库和数据结构更侧重于方便编程和源代码管理。

但是，MySQL的体系结构体现了其设计者的智慧，众多井然有序的子系统和谐地构成了一个高效而又可靠的数据库系统。本章前面的内容所提到的所有技术都可以在这个系统里找到。负责具体实现这些技术的子系统设计和实现得非常优秀，构思巧妙的源代码随处可见。许多有成就的C和C++程序员都对MySQL源代码的精巧和简洁赞叹不已。我经常一边为看不懂那么复杂的源代码而郁闷，一边为它们精巧的设计感到佩服。事实上，就连那些代码的原作者本人也承认，有很多代码是他们灵光一闪时写出来的，他们自己也得经过认真的分析才能把它们弄明白。如果你看懂了这样的源代码，就一定会惊叹它流畅的运行和简洁的结构。

注解 对某些人来说，MySQL系统不容易学习，在遇到问题时也不容易诊断。但只要你掌握了MySQL的体系结构和源代码的精妙之处，这个系统还是非常容易入门的。我个人认为，MySQL有可能成为从事与数据库有关的各种实验的最佳平台。

总而言之，MySQL的体系结构和源代码不适合刚入门的C++新手。如果你想重新考虑编写源代码，就应该继续读下去；我将引导大家进入MySQL源代码的世界。在这之前，我们先来看看这个系统的体系结构。

2.4.1 MySQL 系统体系结构

MySQL的体系结构是一个由多个子系统构成的层次化系统。虽然源代码没有被编译成一系列组件或模块，但那些子系统的源代码却是按照一种可以把各子系统封装在源代码里的层次结构组织的。绝大多数子系统依赖于底层的库（如线程控制、内存分配、网络连接、日志和事件处理以及访问控制等）。基础库、建立在那些库上的子系统以及从其他子系统建立的子系统合起来，构成了一个抽象的API，称为MySQL的C Client API。这个功能强大的API使MySQL系统既可以被用作一个独立的服务器，也可以被用作大型应用程序中的一个嵌入式数据库系统。

MySQL的体系结构封装着SQL接口、查询解析器、查询优化器和查询执行引擎、缓存/缓冲机制以及一个插件式存储引擎。图2-4给出了MySQL的体系结构和它的各子系统。在图2-4的顶部是一些用来与客户端应用程序建立连接的数据库接口。正如大家看到的那样，MySQL为你们所能想到的任何一种编程环境都提供有相应的数据库接口。在图的左边，按系统管理和服务控制分类列出了许多辅助工具，对这些系统管理工具和服务控制工具的详细讨论请参阅Michael Kruckenberg和Jay Pipes合著的*Pro MySQL*^①，该书对MySQL系统管理方面的每一项工作都做了详细的探讨。

^① 由Apress公司2005年出版。

从数据库接口往下一级是连接池（connection pool）层，该层负责处理与用户访问有关的各种用户登录、线程处理、内存和进程缓存需求。连接池的下一层是MySQL数据库系统的核心，这里是对查询进行分析和优化的地方，也是对文件访问进行管理的地方。再往下是插件式存储引擎层，这一层是MySQL的体系结构与众不同的部分之一。插件式存储引擎层使得MySQL系统可以灵活地适应各种数据或文件的存储和检索机制。这种灵活性是MySQL独有的，目前市场上的其他数据库系统都不具备这种通过提供多种数据存储机制而调整数据库的能力。

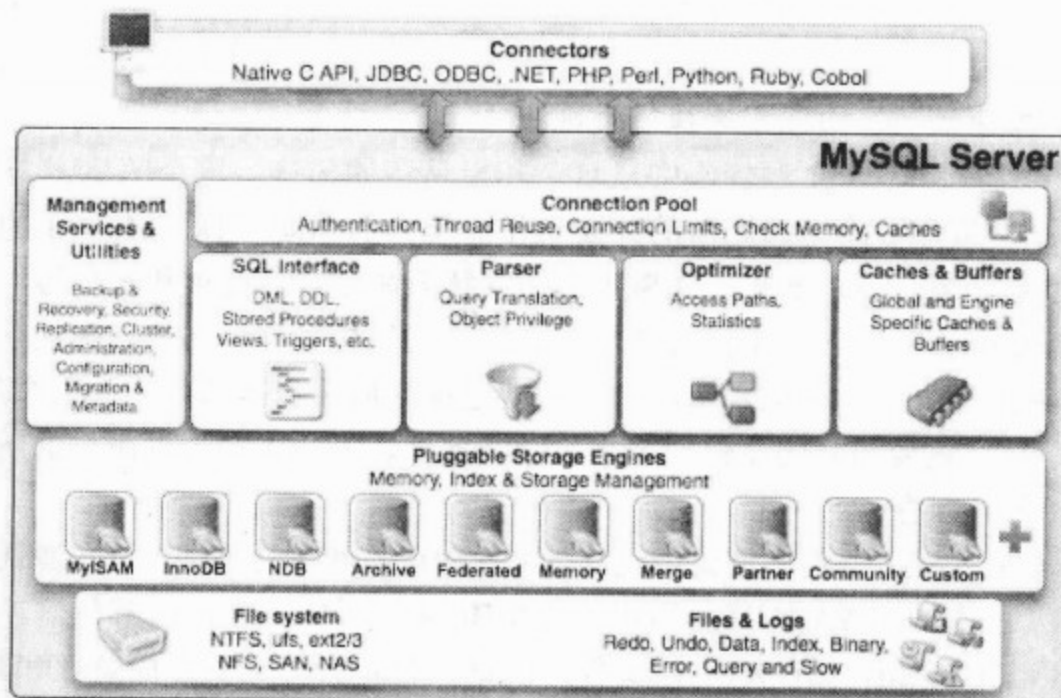


图2-4 MySQL服务器的体系结构（MySQL AB公司授权刊载）

注解 插件式存储引擎最早见于MySQL 5.1版。

插件式存储引擎的下面是MySQL系统的最底层，即文件访问层。这一层是存储机制读/写数据，系统读/写日志和事件信息的地方。这一层也是最贴近操作系统的层，与线程、进程和内存控制有关的操作都发生在这里。

我将按照从客户端应用程序到数据库、再从数据库回到客户端应用程序的流向对MySQL的体系结构进行讨论。在客户端连接器（ODBC、JDBC、CAPI，等等）把SQL语句传递到服务器的时候，首先遇到的将是SQL接口。

2.4.2 SQL 接口

SQL接口提供了从用户接收命令并把结果返回给用户的机制。MySQL的SQL接口是按ANSI SQL标准建立的，与ANSI标准兼容的其他数据库服务器所支持的基本SQL语句都可以在这里使用。虽然在MySQL支持的SQL命令当中有许多带有不属于ANSI标准的选项，但MySQL的开发者在遵守ANSI SQL标准方面做得非常好。

MySQL服务器从网络接收连接请求并为每个连接创建一个线程。线程是MySQL服务器进行查询处理的核心。MySQL是一个真正的多线程应用程序，每个线程在执行时都独立于其他线程（但有几个

特定的辅助线程不在此列)。接收到的SQL命令将存入一个类结构(class structure), 查询结果通过把有关数据写到网络通信协议上传输回客户端。创建出一个线程之后, MySQL服务器将开始解析SQL命令并把解析出来的各个部分保存到一个内部数据结构里去。

2.4.3 解析器

当收到客户端发出的查询并为之创建一个新线程之后, SQL语句将被传递到解析器接受语法验证(或因错误而被拒绝)。MySQL的解析器是用一个很长的Lex-YACC脚本实现的, 用Bison对该脚本进行编译就得到了这个解析器。解析器将构造一个用来在内存里代表查询语句(SQL)的查询结构, 这个树状结构(也称为抽象语法树)可以用来执行查询。

有许多人认为解析器是MySQL源代码中最复杂和最精巧的部分。这个解析器是用Lex和YACC实现的, Lex和YACC原本是为了简化编译器的开发工作而推出的辅助工具。MySQL的开发者使用这些工具建立了一个词法解析器, 它能够把一条SQL语句分解为命令字、选项和参数等一系列最基本的语法元素(记号)并把这些记号存入一个由变量和列表构成的结构。这个结构(它的名字或者说类型是Lex)就是SQL查询的内部表示。也就是说, 查询处理过程的后续步骤都要用到这个结构。Lex结构包含将要用到的表的清单、将要访问的文件的名字、联结条件、表达式和查询命令的所有其他元素, 它们与原始的SQL语句分开保存。

MySQL的解析器读入SQL语句并把该表达式(由各种记号和符号构成)与在源代码里定义的规则进行比较。这些规则是用Lex和YACC代码写出, 再由Bison编译成一个词法解析器。如果你们看过这个解析器的C语言代码(见/sql/sql_yacc.cc文件)的话, 肯定会为它的言简意赅和数量巨大的switch开关分支语句而惊讶不已^①。了解这个解析器的更好办法是阅读尚未经过编译的Lex和YACC代码(见/sql/sql_yacc.yy文件)。这个文件包含着用YACC代码写出来的规则, 相对比较容易阅读和理解。利用Lex和YACC来实现MySQL解析器的做法体现了MySQL AB公司的开放源代码理念: 既然有Lex、YACC和Bison这些专用的编译器开发工具可以利用, 为什么还要创建你自己的语言处理器呢?

在解析器标识该正则表达式并把查询语句分解成一系列基本元素之后, 它将把适当的命令类型分配给相应的线程结构并把控制权返回给命令处理器(command processor, 它有时被认为是解析器的一个组成部分, 但更准确地说是主代码的一部分)。命令处理器其实就是一个长长的switch语句, MySQL所支持的每一种命令分别对应着其中的一个分支。查询解析器只检查SQL语句的语法正确性。它不检查有关的表或属性(字段)是否存在, 也不检查语义错误(比如使用了一个统计函数, 但没有写出必要的GROUP BY子句的情况)。这些事情将由优化器去检查。解析器生成的查询结构将被传递到查询处理器, 从那开始的后续工作将由查询优化器负责控制。

Lex和YACC

Lex的含义是词法解析器生成器(lexical analyzer generator), 它是一个用来把语句/表达式分解成一系列最基本的记号和常数的解析器, 也可以完成一些语法检查工作。YACC的含义是另一个编译器的编译器(yet another compiler compiler), 可以用来标识和处理语言的语义定义。这两个工具

^① Kruchtenberg和Pipes把阅读那些代码的经历比喻为一次洗脑。那些代码非常简洁, 但对不熟悉YACC的人来说恐怕是一个巨大的挑战。

再加上Bison（一个YACC编译器），可以帮助人们简便快速地开发出一个能够分析和处理语言命令的子系统。MySQL就是这样使用这些技术的。

提示 如果你打算为MySQL增加一些新的SQL命令，sql_yacc.yy、sql_lex.h和lex.h文件就将是你的出发点。本书将在第8章对这些文件做更详细的讨论。

2.4.4 查询优化器

有些人认为MySQL的查询优化器子系统应该使用另一个名字。这个优化器使用了一种“选取-投影-联结”策略来处理查询，即先根据有关的限制条件进行选取（SELECT操作）以减少将要处理的元组的个数，再进行投影（对应于关系代数里的投影操作）以减少被选取元组里的属性（字段）的个数，最后根据联结条件生成最终的查询结果。MySQL的查询优化器算不上是最复杂的，它采用的“选取-投影-联结”策略属于一种启发式查询优化机制。它使用的规则很简单：

- 通过计算WHERE子句里的表达式来横向排除多余的数据。
- 只保留在属性（字段）清单里列出的，以及在最后执行联结子句时还需要用到的属性（字段），其他数据全部排除。
- 根据联结条件生成最终的查询结果。

由这几条规则构成的策略可以保证数据检索动作的效率达到或非常接近最优。实践证明，这种“选取-投影-联结”策略即使对于事务处理中的典型查询也能保证很高的效率。图2-5给出了MySQL的查询处理流程。

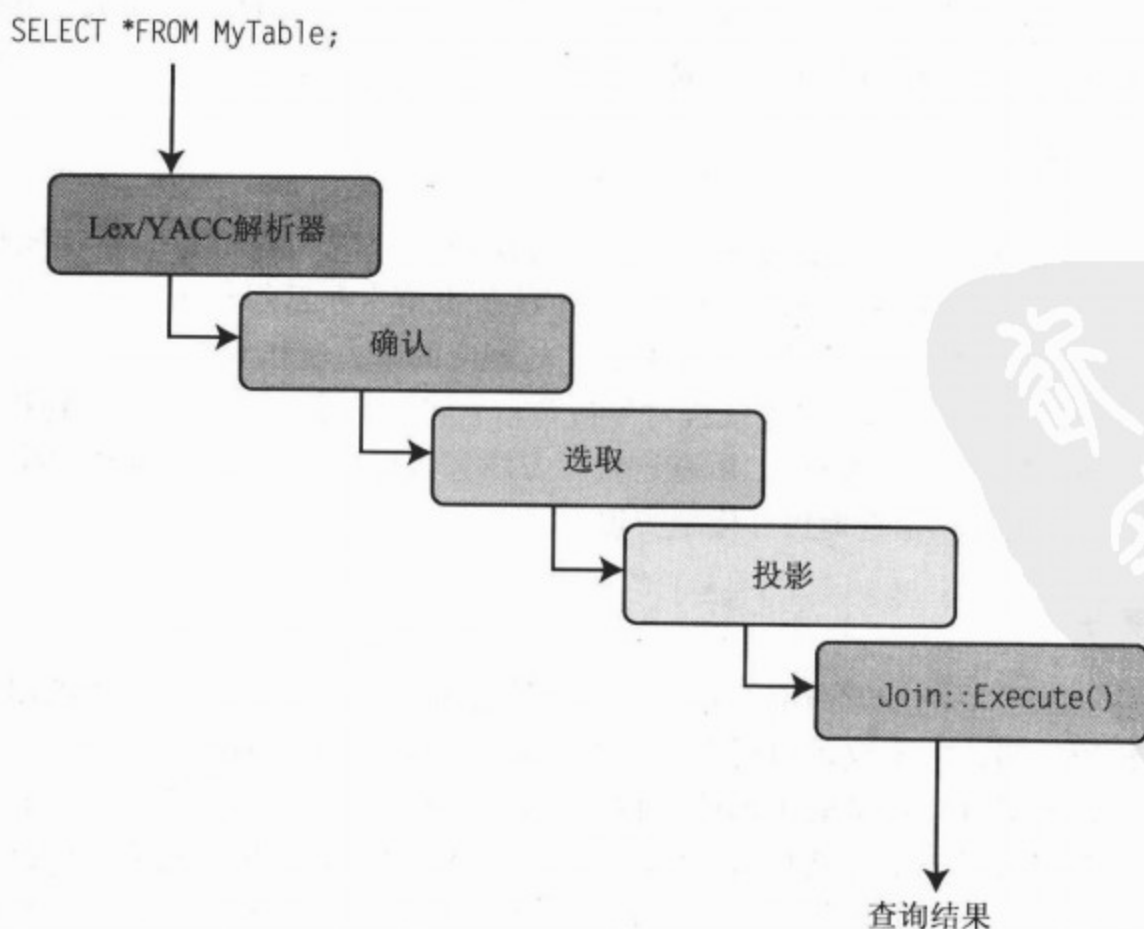


图2-5 MySQL的查询处理流程

注解 这里体现了MySQL AB公司的另一个理念：优化团体的当前需求。

优化器里的第一步是检查有关的表是否存在以及用户是否有访问权限；如果发现错误，返回一条相应的出错信息并把控制权交还给线程管理器（thread manager，也就是负责监听连接请求的listener守护进程）。一旦找到了正确的表，就打开并应用适当的并发控制锁。

在所有的维护和初始化任务全部完成后，优化器将使用内部查询结构（即Lex结构）对查询命令中的WHERE条件进行处理（“选取”操作），其结果作为临时表返回供后续步骤做进一步处理。如果查询命令里有UNION操作符，优化器将以循环方式执行完所有的SELECT操作之后再进行下一步。

优化器里的下一步是执行投影操作。投影操作的执行情况与限定操作很相似，这一步的中间结果仍保存在临时表里，且只保存那些由SELECT语句中的列规范所指定的属性。最后，检查Lex结构里有没有用联结类（join class）给出的JOIN条件；如果有，调用join::optimize()方法。在这个阶段，查询将接受以下优化：对条件表达式进行简化和求值，消除死分支或恒真/恒假条件（以及许多其他类似的优化）。总之，在执行联结操作之前，优化器会尽量减少查询命令里的条件（能简化的简化，能求值的求值，无效的尽量消除）。做出这种安排的原因是因为，联结操作是所有关系操作中开销最大和耗时最多的。还有一点请大家特别注意：不管是不是真的有联结条件，凡是带有WHERE或HAVING子句的查询都要经历联结优化步骤。这样开发者可以把所有的表达式求值代码集中到一个地方。在联结优化步骤完成之后，优化器将通过一系列条件语句调用一系列适当的库方法去执行查询。

查询优化器和执行引擎大概是MySQL源代码中第二个难点，这与它采用的“选取-投影-联结”策略不无关系。更让人头疼的是，MySQL服务器的这个部分是用C和C++代码混合编写的，基本的“选取”操作被编写成一些C方法，联结操作则被编写成一个C++对象。第11章将向大家介绍如何编写自己的查询优化器并用它来代替MySQL自带的优化器。

2.4.5 查询的执行

查询的具体执行是由一组库方法实现的，每种方法负责完成一种特定的查询操作。比如说，mysql_insert()方法负责插入数据，mysql_select()方法负责检索和返回与WHERE子句相匹配的数据。这些库方法分散在许多个源代码文件里，文件的名称与相应的库方法相同或接近（如sql_insert.cc、sql_select.cc）。所有这些方法都有一个线程对象参数，它可以让那些方法直接找到和访问有关的内部查询结构（Lex结构），加快执行速度。所有这些库方法都将使用网络通信路径库返回其执行结果。查询执行库的方法是按照解释型查询执行模型实现的。

2.4.6 查询缓存

虽然不是它自己的子系统，但查询缓存却是查询优化和执行子系统不可缺少的组成部分。查询缓存是一项了不起的发明，它不仅可以缓存查询结构，还可以缓存查询结果本身。如果某个查询的结果就在缓存里，系统就可以直接取出那些数据返回给客户端而跳过整个查询优化和执行阶段；这对那些使用频繁的查询来说效果尤其显著。这项技术也是MySQL独有的；其他的数据库系统只缓存查询，不缓存查询结果。查询缓存还必须能够处理查询结果变“脏”的情况，即数据在上次查询之后有一些发生了变化（比如对基表运行INSERT、UPDATE或DELETE命令的情况），以及时不时地对被缓

存的查询进行清理。

提示 查询缓存是默认打开的。如果你想关闭查询缓存功能，可以使用SQL_NO_CACHE SELECT选项：
SELECT SQL_NO_CACHE id, lname, FROM myCustomer;。

如果你还不熟悉这项技术，可以自己去体验一下：找个记录足够多的表执行一个足够复杂的查询，比如一个JOIN或复杂的WHERE子句。记下它的执行时间，然后再次执行这个查询。请注意时间上的差异，就可以看出查询缓存的作用。代码清单2-1演示了这个练习。

代码清单2-1 MySQL的查询缓存机制的作用

```
mysql> SELECT SQL_NO_CACHE professionals.last_name,
certifications.certificate_level
FROM professionals JOIN certifications
ON professionals.unique_no = certifications.unique_no
WHERE professionals.med_class > 1 AND certifications.last_name = 'Bell';
```

```
+-----+-----+
| last_name | certificate_level |
+-----+-----+
| BELL      | P                |
| BELL      | S                |
| BELL      | Y                |
| BELL      | P                |
| BELL      | S                |
+-----+-----+
5 rows in set (1.94 sec)
```

```
mysql> SELECT SQL_CACHE professionals.last_name,
certifications.certificate_level
FROM professionals JOIN certifications
ON professionals.unique_no = certifications.unique_no
WHERE professionals.med_class > 1 AND certifications.last_name = 'Bell';
```

```
+-----+-----+
| last_name | certificate_level |
+-----+-----+
| BELL      | P                |
| BELL      | S                |
| BELL      | Y                |
| BELL      | P                |
| BELL      | S                |
+-----+-----+
5 rows in set (0.61 sec)
```

```
mysql> SELECT SQL_CACHE professionals.last_name,
certifications.certificate_level FROM
professionals JOIN certifications
ON professionals.unique_no = certifications.unique_no
WHERE professionals.med_class > 1 AND certifications.last_name = 'Bell';
```

```

+-----+-----+
| last_name | certificate_level |
+-----+-----+
| BELL      | P                 |
| BELL      | S                 |
| BELL      | Y                 |
| BELL      | P                 |
| BELL      | S                 |
+-----+-----+
5 rows in set (0.61 sec)

```

```
mysql>
```

2.4.7 缓存和缓冲区

缓存和缓冲区子系统负责保证使用频率最高的数据（或结构，就像你们将看到的那样）能够以最有效率的方式被访问。换句话说，使用频率最高的数据必须随时驻留在内存里待用。因为那些数据随时都驻留在内存里，不需要访问磁盘就可以检索到，所以缓存机制可以大大缩短对那些数据的请求响应时间。MySQL的缓存子系统把所有的缓存和缓冲区功能封装在一组松散搭配的库函数里。那些库函数散布在多个不同的源代码文件里，但我们必须把它们当作同一个子系统的组成部分来看待。

这个子系统实现了很多缓存，绝大多数缓存机制使用的是相同或相似的概念：把数据封装为某种结构，再把这些结构保存为一个链表（linked list）。在MySQL的源代码里，与缓存和缓冲区有关的代码不是集中在某个地方而是散布在各地，哪些地方需要用到它们，它们就会出现在哪里。下面我们来看一下这些缓存。

1. 表缓存

表缓存（table cache）是为了最大限度地减少打开、读取和关闭表（磁盘上的.FRM文件）的开销而创建的。因此，表缓存的主要用途就是把关于表的元数据保存在内存里。这可以大大加快有关线程读取表结构信息的速度而无需每次都打开文件。每个线程都有它自己的表缓存结构表，因而可以独立于其他线程对表进行访问，即使某个线程改变了某个表的模式（但还没有提交那些修改），另一个线程可以继续按照原来的模式使用该表。这种结构是一个封装表的所有元数据信息的简单结构，这些结构在内存里以链表的形式存放并与各有关线程相关。

2. 记录缓存

记录缓存（record cache）是为了加快存储引擎的顺序读性能而创建的。因此，记录缓存通常只用在表扫描期间。它就像是一个预读缓冲区，每次检索一个数据块，从而减少扫描期间的磁盘访问次数。一般来说，磁盘访问次数越少，就意味着性能越高。有意思的是，MySQL在顺序写数据的时候也要用到记录缓存：先把新数据（或修改后的数据）写入记录缓存，等它写满时再把全部数据写到磁盘上。这样一来，写性能也得到了改善。因为记录缓存可以大幅提高顺序读/写（称为引用的局域性）的性能，所以它最经常与MyISAM存储引擎（但并非仅限于此）配合使用。记录缓存是以一种不可知的方式实现的，与用来访问存储引擎API的代码互不干扰。因为记录缓存是在API各层的内部实现的，所以程序员无需采用任何额外步骤就可以享受到记录缓存的好处。

3. 键缓存

键缓存 (key cache) 其实就是一个用来缓存索引数据的缓冲区, 其内容是一个来自索引文件 (B-树) 的数据块; 只有 MyISAM 表 (磁盘上的 .MYI 文件) 才使用键缓存。索引本身被封装在键缓存结构里, 这些结构在内存里被存储为一个链表。在第一次打开一个 MyISAM 表的时候, 系统会为它创建一个键缓存。此后的每一次索引读操作都将先访问这个键缓存。如果在缓存里找到了需要的索引, 则从那里读出它; 如果没有找到, 就将从磁盘读入一个新的索引块放到缓存里。这个缓存的长度有一个默认值, 但你可以通过配置变量 `key_cache_block_size` 来修改。一般来说, 某个索引文件里的所有数据块是不可能同时容纳在内存里的。那么, 系统又是如何知道哪些索引块已经被使用过了呢?

缓存有它自己的监控机制, 该机制随时记录各索引块的使用频率。键缓存用来记录各索引块的“热度”。在这里, 热度指索引块被使用了多少次。变量 `warm` 可取的值包括 `BLOCK_COLD`、`BLOCK_WARM` 和 `BLOCK_HOT`。当某个索引块变“冷”, 另一个索引块变“热”时, 后者就会被读入键缓存而取代前者。这种做法其实是一种“最近最少使用” (least recently used, LRU) 页面交换策略, 与操作系统中的虚拟内存管理和磁盘缓冲机制所使用的算法是一样的。实践证明, LRU 策略相当高效, 即使与更复杂的页面交换算法相比也毫不逊色。键缓存用类似的方法随时跟踪各索引块是否变“脏” (被修改过)。当一个“脏”索引块被交换出内存时, 它的数据将被写入磁盘上的索引文件。如果被清除的是一个“干净”的索引块, 只要从内存里删除它就完事了。

注解 经验表明, LRU 算法至少在 80% 的情况下都能取得最优的效果。在一个追求速度以及简单就意味着可靠的世界里, 能在 80% 的情况下取得最优效果的解决方案已经很了不起了。

4. 权限缓存

权限缓存 (privilege cache) 用来存放用户账户的授权数据。这些数据与访问控制表 (access control list, ACL) 的存储方式是一样的, 它列出了某个用户对系统里某个对象的全部权限。权限缓存把用户的每项权限封装在一个结构里, 这些结构在内存里被存储为一个“先进后出” (first in-last out, FILO) 的散列表。权限缓存里的数据是在用户登录上机和初始化期间从各有关权限表里读出并集中到权限缓存里的。

5. 主机名缓存

主机名缓存 (hostname cache) 是另外一种辅助性的缓存机制, 与权限缓存很相似。它也被实现为一种以结构为元素的栈。这个缓存里的内容是与服务器相连接的所有主机名。你或许会感到惊讶, 但主机名的使用频率确实非常高, 是专用的缓存的候选机制。

6. 其他缓存机制

MySQL 的源代码里还有许多随处可见的小缓存机制, 用在复杂的联结操作里的联结缓冲区 (join buffer cache) 就是其中的一个例子。比如说, 有些联结操作需要把第 1 个表的一条记录与第 2 个表里的所有记录进行比较。在这类场合, 用一个缓存来存储那些记录可以大大加快联结操作的速度而无需反复多次地把第 2 个表里的记录读入内存。

2.4.8 通过插件式存储引擎访问文件

支持多种存储引擎, 或者说支持多种文件类型的能力是 MySQL 的最佳功能之一。这使得数据库专

家可以根据具体应用程序的需要为他们的数据库选择一种性能最佳的存储引擎，比如说，为用于事务处理的数据库选用具备事务控制能力的存储引擎，为读取频繁但很少被修改的表（例如一个字典表）选用内存存储引擎，等等。

MySQL AB公司在第5版里增加了一项新的体系结构设计，这项设计使得增加新存储类型的工作变得更容易了。这个新机制叫作MySQL插件式存储引擎。MySQL AB公司已尽了最大努力让服务器能够通过插件式存储引擎得到扩展。插件式存储引擎的核心是文件访问层的一个抽象接口，任何人都可以利用这个API接口去建立新的文件访问机制，MySQL AB公司称之为“存储引擎”（storage engine）。这个API提供了一套用于读/写数据的方法和访问工具。这些方法共同构成了一个标准化的模块化体系结构，不同的存储引擎可以调用同样的方法去访问数据库里的数据（所谓“插入式”存储引擎的意思是指不同的存储引擎都可以使用同样的API“插入”服务器）。

插件式存储引擎最让人感兴趣的地方是允许你在一个给定的数据库里为每个表指定一个不同的存储引擎，你甚至可以在创建一个表之后改变它的存储引擎。这种灵活性和模块化使得数据库的实现者可以随时根据需要创建新的存储引擎。下面这条命令可以用来改变某个表的存储引擎：

```
ALTER TABLE MyTable  
ENGINE = InnoDB;
```

插件式存储引擎大概是MySQL最独特的功能了。其他品牌的数据库系统没有一个能像MySQL这样在文件访问层具有如此大的灵活性和可扩展性。在接下来的几个小节里，将对MySQL服务器支持的所有存储引擎进行描述并对如何创建自己的存储引擎做一个简要介绍。本书将在第7章向大家演示如何创建你自己的存储引擎。

各类存储引擎的优缺点为数众多，且各不相同。比如说，在MySQL里，有几种存储引擎支持并发处理。MySQL的默认存储引擎是MyISAM。它支持表级的并发控制锁：当某个进程对某个表进行更新操作时，在这个操作完成之前，其他进程将不能访问该表里的任何数据。在MySQL所支持的存储引擎当中，MyISAM存储引擎是最快的，这是因为MyISAM表都按照ISAM原则（indexed sequential access method，意思是“索引过的顺序访问方法”）进行过优化。BDB（Berkley Database的简写）表支持页面级的并发控制锁：当某个进程对某个表进行UPDATE操作时，在这个操作完成之前，其他进程将不能访问与被刷新的记录同处于一个内存页面里的任何数据。InnoDB表支持记录级（也叫作“行级”）的并发控制锁：当某个进程对某个表进行UPDATE操作时，在这个操作完成之前，其他进程将不能访问被更新的记录（行）。既然如此，在并发访问（尤其是数据更新操作）比较频繁的系统上就最适合使用InnoDB表类型。不过，这些存储引擎在只读环境下（例如Web服务器或查号台之类的应用）的性能都很不错。

在数据库系统里，我们刚才讨论的并发操作是用一些专用的命令实现的，这些命令构成了所谓的事务子系统。目前，MySQL只提供了三种支持事务的存储引擎：BDB、InnoDB和NDB。事务提供了一种把一组操作当作一个原子化的操作来执行的机制。比如说，银行转账系统里的数据库应该把从一个账户向另一个账户汇款的两步操作（把钱从第一个账户取出、再放入第二个账户）当做一个整体来执行，在此期间不允许出现意外。事务机制将把这两步操作封装为一个原子化的操作，如果在所有操作完成之前遇到了问题，就会撤销已经做出的任何修改，从而避免第一个账户里的余额减少了、但第二个账户的余额没有增加的情况。下面是一组用来完成上述事务的SQL语句，其中的第一条和最后一条语句是事务命令。

```
START TRANSACTION;
UPDATE SavingsAccount SET Balance = Balance - 100
WHERE AccountNum = 123;
UPDATE CheckingAccount SET Balance = Balance + 100
WHERE AccountNum = 345;
COMMIT;
```

在实践中，在追求速度的场合，绝大多数数据库专家会选择MyISAM表类型；在需要事务支持的场合，他们会选择InnoDB。MySQL允许你为同一个数据库里的每一个表分别指定一种表类型。事实上，同一个数据库里的表不必是同样的类型。这种存储引擎的多样性让MySQL数据库系统具备了更广泛的适用性。

有意思的是，MySQL还允许你编写自己的存储引擎。为了让系统开发人员熟悉这项功能，MySQL准备了许多例子和代码框架。允许人们自行编写存储引擎的能力使得为MySQL添加对复杂的、专利数据格式和访问层的支持不再只是一种设想。

1. MyISAM

MyISAM存储引擎是MySQL的默认文件访问机制，在创建时没有在CREATE语句里明确设置ENGINE选项的所有表都将使用这种存储引擎。绝大多数LAMP应用、数据仓库、电子商务和企业管理应用使用的都是这种存储引擎。MyISAM在ISAM的基础上增加了一些新的缓存和索引机制。为了提高访问速度，这类表还普遍使用了数据压缩和索引优化技术。此外，MyISAM存储引擎还为并发操作准备了表级的锁定机制。MyISAM存储机制的优点是可靠性高、适用范围广、数据检索速度快。在强调数据检索速度（读性能）的场合，MyISAM是首选的存储引擎。

提示 可以通过修改服务器配置变量STORAGE_ENGINE的办法来改变MySQL的默认存储引擎。

ISAM

ISAM文件访问方法已经出现很久了。ISAM最初是由IBM公司开发的，后来用在了System R里。（System R是IBM公司推出的第一代RDBS，很多人认为它是目前市场上所有RDBS的鼻祖和参照物。但也有人认为Ingres才是最早的RDBS。）

在存储数据的时候，ISAM将把数据组织为一些固定长度属性的元组。那些元组按照一种给定的顺序存储，而这是为了加快从磁带访问数据的速度。是的，磁带曾经是数据库实现者在存储介质方面的唯一选择，除非你想使用穿孔卡片。我每次提起我用过穿孔卡片的时候都会有一种廉颇老矣的感觉。如果你也记得穿孔卡片，那你也许还能分享一个现在已很少有人体会过的感受——手里拿着一摞卡片奔波于实验室之间。

ISAM文件还有一个外部的索引机制，该机制通常被实现为一个以指针（磁带块编号和计数值）为元素的散列表，可以让你把磁带快进到想到达的位置。这可以加快对存储在磁带上的数据的访问速度——到底多快要取决于磁带的快进速度。

ISAM机制是为磁带而创建的，但完全可以用于磁盘文件系统（事实也正是如此）。ISAM机制的最大优点是索引非常小——绝大多数ISAM表的索引可以全部放在内存的索引缓存区里，所以搜索起来速度非常快。ISAM机制的某些更为后期的版本允许我们为表创建多个索引，而这意味着可

以使用多种搜索机制去访问有关的文件（表）。这种外部索引机制已经成为所有现代数据库存储引擎的标准做法。

MySQL的早期版本都有ISAM存储引擎（那时作为表类型来引用），但这个ISAM存储引擎现在已经被MyISAM存储引擎取代了。据说，MySQL AB公司已经在考虑把MyISAM存储引擎升级换代为一种更现代的事务存储引擎。

注解 MySQL的早期版本都支持ISAM存储引擎。自MyISAM出现以后，MySQL AB公司逐步淘汰了ISAM存储引擎。

2. InnoDB

InnoDB是一个采用GNU公共许可证（GNU Public License, GPL）发行的第三方存储引擎，它来自Innobase公司（www.innodb.com）。InnoDB几乎总是用在需要支持事务处理的应用里。InnoDB支持传统的ACID事务处理原则（请参见下面的文本框内容）和外键约束机制。InnoDB里的所有索引都采用了B-树结构——把索引记录存放在叶结点里。InnoDB改进了MyISAM的并发控制，可以提供行级的锁定。在强调可靠性和需要支持事务处理的场合，InnoDB是首选的存储引擎。

什么是ACID

ACID是atomicity（原子性）、consistency（一致性）、isolation（隔离性）和durability（耐久性）等4个英文单词的首字母缩写。ACID原则是数据库理论中最重要的概念之一，它定义了一个能可靠地用在事务处理环境里的数据库系统应该达到哪些要求。

- 原子性。对于一个包含着多个命令的事务，数据库必须能够做到让那些命令对数据做出的改动要么全部成功，要么什么都不做。换句话说，每个事务都是原子化的。只要其中一条命令没有成功，整个事务就将失败，此前已经做出的修改必须全部撤销。这一点对那些运行在高度依赖事务处理的环境（比如证券市场）里的系统来说尤其重要。以银行账户之间的转账为例，把钱从一个账户转入另一个账户通常需要好几步操作。如果把钱从第一个账户转出之后因为某种原因没能把钱转入第二个账户（事务失败了），就必须把钱退回给第一个账户，否则就会引起顾客的愤怒。也就是说，整个事务要么所有命令都执行成功，要么什么都不做。
- 一致性。只有合法且有效的数据才能进入数据库。如果某个事务里有一条命令违反了这项原则，就必须放弃整个事务并把有关数据恢复到开始执行这个事务之前的样子。反过来讲，如果某个事务成功了，它将以一种符合数据库一致性规则的方式改变有关的数据。
- 隔离性。如果有多个事务在同时执行，它们必须做到不相互干扰。这项原则显然是对数据库系统的并发控制机制提出的要求。一个可靠的数据库系统必须保证任何一个事务都不会影响（修改、删除，等等）其他事务正在使用的数据。有好几种办法可以实现这一目标，其中最常用的办法是使用某种锁定机制来阻止其他事务去访问前一个事务正在使用的数据，直到前一个事务结束为止。请注意，隔离性原则并没有规定哪个事务应该优先执行，只要求数据库系统必须做到任意两个事务不能相互干扰。
- 耐久性。任何事务都不应该导致数据丢失，这既包括现有的数据，也包括在事务过程中被创

建或改变的数据。这种耐久性通常由健壮的数据备份和恢复功能负责提供和保证。有些数据库系统使用日志来保证任何尚未提交的数据在系统重启时会自动得到恢复。

3. BDB

BDB是Berkley Database的缩写。BDB是一个来自Sleepycat公司 (www.sleepycat.com) 的第三方存储引擎。BDB存储引擎被认为是InnoDB的更新换代产品, 支持事务以及COMMIT和ROLLBACK等额外的事务功能。BDB支持散列表、B-树、简单的基于记录编号的存储机制和永久查询。

注解 Oracle现在虽然拥有InnoDB和BDB, 且在接下来的几年也将保持两种技术共存的状态。但是在不久的将来, BDB也许会成为不被支持的存储引擎。

4. 内存

内存存储引擎 (memory storage engine) (有时被称为HEAP表) 是一种驻留在内存里的表, 它使用了一种散列机制来加快常用数据的检索速度。这种表要比存储在磁盘上的表快很多。它们在使用上与其他存储引擎没有什么不同, 只不过数据是存储在内存里并只在MySQL会话期间有效而已。在系统关机 (或崩溃) 时, HEAP表里的数据将丢失。内存存储引擎特别适合用来存储那些访问频繁但很少需要修改的静态数据, 比如邮政编码、国家、地区、产品目录之类的字典表等。HEAP表还可以用在那些利用快照技术提供分布式或历史数据访问服务的数据库。

提示 基于内存的表将被创建在/data_dir/database_name/table_name.frm子目录下。利用启动选项--init-file=file可以在系统开机启动时自动创建出基于内存的表来。在这种情况下, 给定文件应该包含用来重新创建那些表的SQL语句。因为表只需创建一次, 并且表的定义在系统重启时也不会被删除, 所以你可以省略CREATE语句。

5. 合并

合并存储引擎 (merge storage engine) 是用一组有着相同结构 (元组布局或模式) 的MyISAM表建立的, 其效果是那些表可以被当作一个单个的大表来使用。那些表将按照它们各自的位置来分区, 并不需要使用额外的分区机制。所有的表必须驻留在同一台机器上 (通过同一个服务器访问)。用户只需使用单个操作或SELECT、UPDATE、INSERT和DELETE等语句就可以访问到那些表里的全部数据。幸好, 对合并表发出的DROP命令只解除那些表的合并关系, 不会改变那些最初的表。

这种表类型的最大优点是速度。利用合并存储引擎, 可以把一个大表分成几个较小的表并保存在不同的磁盘上, 再通过一些合并表规则把它们合并起来以便同时对它们进行访问。因为每个表里的数据只是全部数据的一部分, 所以搜索和排序操作的执行速度会快很多。比如说, 如果你是按照某种特定条件来划分数据的, 就可以只搜索与你想要搜索的数据有关的各个子表而不是去搜索一个体积庞大的表。类似的, 对表进行修复的工作也将变得比较容易, 因为修复几个较小的文件要比修复一个大文件要容易和迅速得多。据推测, 绝大多数错误都集中发生在一两个文件里, 不需要重新创建或修复所有的数据。但这种存储引擎有以下几个缺点:

- ❑ 你只能使用结构相同的MyISAM表来构成一个大的合并表。这就把合并存储引擎的应用范围限制在了MyISAM表上。若是合并存储引擎可以接受任何一种表类型的话, 合并存储引擎肯定会

比现在更加有用。

❑ 不允许进行替换操作。

❑ 与一个普通的表相比，使用索引去访问一个合并表的效率要低一些。

合并存储机制最适合用在特大型数据库（very large database, VLDB）应用里，比如一个因为数据量太大而需要把数据分散到多个表、甚至是多个数据库里去的数据仓库。

6. 档案

档案存储引擎（archive storage engine）是一种用来把大量数据保存为某种压缩格式的机制。档案存储机制最适合用来存放和检索那些不需要频繁访问的档案性或历史积累性数据。就拿系统的操作日志来说吧，普通用户肯定不需要每天都去检索和使用这种东西，但它们在系统出问题的时候就派上用处了。

档案存储引擎没有提供任何索引机制，唯一的访问办法是扫描整个表。因此，档案存储引擎不适合用于日常的数据库存储和检索操作。

7. 联合

联合存储引擎（federated storage engine）是一种用来从多个数据库系统创建一个表的机制。联合存储引擎的工作情况与合并存储引擎很相似，但它还允许你把来自多个数据库服务器的数据（表）链接在一起。这个机制的主要用途也正是把来自其他数据库系统的表链接起来。联合存储引擎最适合用在分布式环境或数据仓库环境里。

联合存储引擎最吸引人的地方是它不移动数据，也不要求远程的表是同一种存储引擎，联合存储引擎会在存储和检索有关数据的过程中自动完成必要的转换。这充分体现出了插件式存储引擎层的强大威力。

8. 群集/NDB

群集存储引擎（cluster storage engine）（在需要与集群产品相区别的场所称为NDB^①）为MySQL提供了群集服务器的能力。群集存储引擎的基本用途是在一个高可用性和高性能的环境里集中使用多个MySQL服务器提供数据库服务。群集存储引擎不存储任何数据，具体的数据存储和检索操作由群集里的各有关数据库所使用的存储引擎负责执行，群集存储引擎只负责控制如何把数据分布到群集簇中的各个数据库以提供冗余和改善性能。NDB存储引擎同时提供了一个API供人们创建可扩展的群集解决方案。

9. CSV

CSV存储引擎用来创建和读写CSV（comma-separated value，逗号分隔的值）格式的表文件。CSV存储引擎不需要把数据复制为另一种格式，CSV表的元数据将它的文件名一起直接保存在服务器上的数据库文件夹里。CSV存储引擎可以让数据库用户快速方便地使用由电子表格软件生成的结构化商务数据。CSV存储引擎没有提供任何索引机制。

10. 黑洞

黑洞存储引擎（blackhole storage engine）是一项既有趣又非常有用的功能。它允许系统写数据，但并不把写入的数据真正保存起来。不过，如果激活了二进制日志功能，有关的SQL语句将被记载到日志里。这种存储引擎为数据库专家提供了一种临时阻断数据的进出以改变表类型的手段。此外，在

^① 关于NDB API的更多信息请访问<http://dev.mysql.com/doc/ndbapi/en/overview-ndb-api.html>。

只是想测试一下某个应用以确保它是在写数据，而不是想把那些数据真正保存起来的场合，这种存储引擎也非常方便。

11. 定制

定制存储引擎（custom storage engine）可以是你为了改进数据库服务器而自行创建的任何一种存储引擎。比如说，你想创建一个存储引擎来读取XML文件。你当然可以把XML文件转换为表，但如果你有很多XML文件，这么做会相当麻烦。如果你打算为此创建一个定制的存储引擎，下面给出了这一过程的简要概括。

如果你正在考虑使用XML存储引擎来读取一些特定的XML文件，你应该做的第一件事是分析你的XML文件的格式并确定你想怎样解决XML文件自我描述的“问题”（说是问题，这其实是XML文件的基本特性）。我们不妨假设那些XML文件都包含着一些同样的基本数据类型，但也有一些不同的标记，而且那些标记的顺序也不尽相同。在这个例子里，我们不妨假设你决定使用一些样式表把那些XML文件的格式统一起来。

在决定使用什么样的格式之后，就可以参考MySQL源代码提供的存储引擎的例子（可以在MySQL主源代码树的.\storage\example文件夹里找到那些源代码）来开发自己的存储引擎了。你会找到一个制作文件和两个源代码文件（ha_example.h和ha_example.cc），其内容是能够让存储引擎插入系统并运转起来的一套完整的代码框架。因为那些代码只是个框架，所以它们本身没有什么实际的用处，但可以阅读代码里的注释来确定存储引擎需要实现哪些功能以及如何实现。比如说，用来打开文件的方法是ha_example::open。通过分析样板存储引擎的源代码文件，可以在ha_example.cpp文件里找到这个方法。代码清单2-2是open方法的一个例子。

代码清单2-2 打开表的方法

```
/*
  Used for opening tables. The name will be the name of the file.
  A table is opened when it needs to be opened. For instance
  when a request comes in for a select on the table (tables are not
  opened and closed for each request, they are cached).

  Called from handler.cc by handler::ha_open(). The server opens all tables by
  calling ha_open() which then calls the handler specific open().
*/
int ha_example::open(const char *name, int mode, uint test_if_locked)
{
    DEBUG_ENTER("ha_example::open");

    if (!(share = get_share(name, table)))
        DEBUG_RETURN(1);
    thr_lock_data_init(&share->lock, &lock, NULL);

    DEBUG_RETURN(0);
}
```

提示 你也可以在Microsoft Windows环境下创建你自己的存储引擎。在这种情况下，文件位于一个Visual Studio项目中。

代码清单2-2给出的例子解释了`ha_example::open`方法的用途、调用方式和返回值。如果你现在还看不太懂那些源代码，别灰心，反复多读几遍，等你熟悉了MySQL的编程风格就容易领会其中的奥妙了。

注解 早期的MySQL版本（5.1版之前）允许自行创建定制的存储引擎，但你必须重新编译MySQL服务器的可执行文件才能让你做的改动生效。从5.1版开始，新的插入式体系结构和模块化的API为存储引擎提供了更丰富的实现方式和功能，更重要的是使得存储引擎的开发工作可以完全独立于MySQL的系统代码。因此，你不必直接修改MySQL源代码。你的新存储引擎项目允许你创建定制的引擎，然后编译并把它与正在运行的MySQL服务器链接起来。

把示例存储引擎及其工作原理弄明白之后，应该先把那些文件复制到你的工作子目录并重新命名为一个更贴切的名字，然后再开始修改那些文件来读取XML文件。像所有优秀的程序员那样，你应该每次只改写一个方法并在改好后对它进行全面的测试，等它的工作情况完全达到你的要求之后再去修改下一个方法。当想要的功能全部实现出来之后，编译你的新存储引擎并把它链接到正在使用的MySQL服务器，新的存储引擎就可以为用户提供服务了。

这听起来似乎是个相当艰巨的任务，但真的动起手来就会发现其实并不难。这种练习也是研究MySQL源代码的一个好办法。本书第7章将带领大家一步一步地创建新的定制存储引擎。

2.5 小结

本章介绍了典型RDBS的体系结构。虽然不能与一本全面探讨数据库理论的图书相比，但本章内容应该能够让大家对关系数据库的体系结构及其工作流程有一个基本的了解。本章还探讨了MySQL服务器的体系结构并说明了如何找到MySQL服务器的各个组成部分的源代码。

了解RDBS的工作原理MySQL服务器的体系结构，将为你进一步扩展MySQL数据库系统打下良好的基础。有了MySQL体系结构的知识，你就全副武装好了（但不会非常危险）。

下一章将介绍MySQL的源代码并开始学习如何根据自己的需要去扩展MySQL系统。好了，卷起袖子，坐到电脑前；我们要开始修改源代码了！

本章将对MySQL的源代码以及如何获得和构建一个MySQL系统做全面的介绍。此外，本章还将介绍MySQL源代码的内部机制、编程指导原则和如何维护代码的最佳实践。在这些代码当中，将重点讨论负责处理查询的部分，这将为学习第7章及以后的主题打下基础。

3.1 预备知识

本节将向大家介绍一些在修改MySQL源代码时应该注意的原则，以及如何获得那些源代码。在这之前，再说一说MySQL的许可证问题。

3.1.1 了解许可证

在开始修改一个开源软件之前，先要考虑如何使用那些修改。具体地说，你打算如何获得它的源代码并对它做哪些修改？根据你的修改目的，你的选择很可能与其他人完全不同。修改MySQL源代码的目的大致可以分为以下3种：

- ❑ 你想知道MySQL的构造，想试试这本书里的例子或者想自己做一些实验。
- ❑ 你想为本人或你的公司开发一项功能，但没打算把那项功能发布到公司之外。
- ❑ 你正在开发一个应用或扩展模块，并且计划推向市场或与其他人分享。

第1章中曾经讨论过，一名开源开发人员在修改一个使用开源许可证发行的软件时，应该承担哪些责任。因为MySQL同时使用了GPL和一种商业许可证（称为双重许可证），所以我们必须把这两种许可证都允许和不允许使用MySQL源代码做什么弄清楚。下面从GPL开始讨论。

出于纯学术的目的来修改源代码，在GPL许可证下是允许的。GPL明确地把修改源代码并用作实验的自由赋予了你。至于你的成果能否在GPL下发行，那还要看这个成果有多大的价值。比如说，如果你修改出来的代码只适用于少数情况（只适合少数用户的特殊用途），这些代码就可能进入不了源代码库。类似地，如果你的代码纯属学术研究或课程练习，这些代码很可能对你以外的其他人没什么价值。MySQL AB公司一般也不会把你在MySQL源代码里实验各种选项和功能的学术练习看作是给MySQL系统增加功能。可是，如果你的实验最终产生了一项有意义的功能，就有义务把它公诸于世。就本书的目的而言，你修改MySQL的源代码应该以不公开修改为前提。我当然希望你认为这本书里的练习既有实质意义又不乏趣味性，但我并不认为它们已经好到无需进一步开发就可以被吸收到MySQL系统里去的程度。如果你在这些练习的基础上对MySQL做出了重大的改进，请接受我的祝福——还有，别忘了告诉别人你是从本书获得的启发。

如果你是在为你本人或所在的组织修改MySQL源代码，并且不打算与他人共享你的修改，应该购买相应的MySQL Network技术服务套餐。MySQL的商业许可证条款允许你修改（你甚至可以从MySQL AB公司获得帮助）并合法地拥有这些代码。

类似地，如果你修改源代码的目的是为了发布那些改动，根据GPL许可证里的有关条款，你将必须免费发行修改出来的源代码（但GPL允许你收取一些工本费）。还有，GPL许可证既不允许你为你做出的修改申请专利，也不支持你享有那些修改的所有权。总而言之，你有义务把它们公诸于世——如果你选择不由你本人来公开修改，就应该把那些代码无偿捐献给MySQL AB公司，并由他们来决定是否要把那些代码集成到产品里去，而那些代码将成为MySQL AB公司的财产。另一方面，如果你确实想为自己对MySQL的修改申请专利（比如把MySQL集成到你开发的某个嵌入式系统里，或是其他类似的情况），应该在开发工作开始之前与MySQL AB公司探讨你的计划。MySQL AB公司将协助你制定出一个既能满足你的要求，又能保护他们权益的解决方案。

3.1.2 获得 MySQL 源代码

获得MySQL源代码的办法有两种，其一是使用MySQL公司官方使用的源代码控制软件（BitKeeper）去获得最新的版本；其二是撇开源代码控制软件直接下载你想要的某个特定版本的源代码。如果你希望自己做出的修改有机会被集成到MySQL系统里去，应该使用源代码控制软件。如果你只是进行学术研究或不打算公开做出的修改，应该直接从MySQL AB公司的MySQL Network网站或MySQL AB公司的开发园地主页下载源代码。

提示 如果你既不要最新的源代码版本，也不想为MySQL添砖加瓦的话，推荐你从<http://dev.mysql.com>下载MySQL的源代码。

什么是源代码控制

源代码控制（source control，也叫作“版本控制”、“代码储存库”或“源代码树”）是一种把有关文档存放到一个中央位置并追踪记录文档修改情况的机制。版本控制技术通常被实现为一种树结构（或类似的层次化结构），原本是为管理大量的工程图纸和字处理文件而开发的。利用这类技术来管理源代码也非常有效——程序员可以随时保存和检索源代码，修改并重新存回储存库；人们把这个过程形象地比喻为签入（check in）和签出（check out）。

源代码控制不仅可以通过管理有关的文件来管理源代码，还可以方便地追踪文件的修改情况。绝大多数源代码控制软件都具备文件分支/融合管理功能，即允许多人同时修改同一个源文件（称为分支），稍后再解决它们之间的冲突（称为合并）。简单地说，源代码控制可以帮助人们管理和追踪储存库里的文件的变化情况。源代码控制是绝大多数系统管理/配置工具箱里的重要组成部分之一。MySQL AB公司使用的源代码控制软件可以先让众多的程序员同时对MySQL源代码的任何一个部分进行开发或修改，然后再决定哪些修改可以进入最终的源代码版本。

如果你已经购买了MySQL Network技术支持套餐，应该请MySQL Network的代表来帮你挑选一个最适合你的源代码版本，并告诉你应该从何处下载。你的MySQL Network代表通常会为你申请一个以

只读权限访问MySQL源代码的登录名和口令。

1. 使用BitKeeper

通过源代码控制软件获得MySQL源代码需要用到一个名为BitKeeper的程序（详见www.bitkeeper.com）。BitKeeper是一种可以让程序员在一个分布式环境里（通过因特网）保存源代码和有关文档的配置管理工具。

注意 BitKeeper存储了MySQL源代码最新版本、分支以及所有其他的中间代码。MySQL AB公司的程序员每天都要使用这个工具来完成日常工作，所以它不可能总是处于最稳定的状态；如果你决定使用BitKeeper，请务必注意这一点。绝大多数新功能要么尚未完成，要么还有待细化。如果你必须选择一个稳定的版本，就应该使用“代码快照”（稍后进行讨论）或某个已正式发布的源代码版本。

需要你做的第一件事就是记住“耐心”这个词。这个过程有点儿繁琐，还很容易出错。虽然大部分人对它的印象还不错，但有些用户在获得和使用BitKeeper客户端程序的过程中确实遇到过或大或小的麻烦。不过，如果你按照我的指示去做的话，应该不会遇到任何问题。

2. 安装BitKeeper

首先，需要下载一份免费的BitKeeper客户端程序。这个客户端程序只能在与POSIX标准兼容的Unix系统上使用，但如果你安装了Cygwin，它也可以在Windows系统上运行。请参见下文标题为“在Windows平台上使用BitKeeper”的文本框内容。在安装客户端程序之后，就可以用它从MySQL的源代码库里下载代码了。在宽带网上，整个过程只需花费几分钟的时间；如果你的上网速度比较慢，下载全部的源代码需要比较长的时间。下载BitKeeper客户程序的具体做法很简单，打开浏览器去访问www.bitkeeper.com/Hosted.html页面即可。

在这个页面上有一个供你下载BitKeeper客户端程序的链接。先单击“Download the client”链接，再选择把bk-client.shar文件保存到你的登录文件夹（你也可以任意选择）里就行了。你下载的BitKeeper客户端程序还没有经过编译，所以还需要把它编译成可执行文件。可以用以下命令完成：

```
%> /bin/sh bk-client.shar
%> cd bk_client-1.1
%> make
```

如果你的系统配置正确并已经安装了gcc和make，BitKeeper客户端程序的编译工作就应该不会有问题。接下来是下载MySQL的源代码树，请输入以下命令：

```
sfioBall -r+ bk://mysql.bkbits.net/mysql-5.1-new mysql-5.1
```

这条命令的意思是让BitKeeper客户程序连接名为mysql-5.1-new的MySQL源代码树，并把文件下载到名为mysql-5.1的新文件夹里。你应该看到一长串信息，表明MySQL的源代码正在被传输到你的系统里。传输工作全部完成后，MySQL源代码最新版本就在你的系统里等你去研究。

提示 可供下载的MySQL源代码树的清单可以在http://mysql.bkbits.net上查到。

上述步骤只能让你获得一份源代码树的副本；但你无法把修改上传到MySQL的源代码库里。如果真的想为MySQL添砖加瓦，还需要一个许可证密钥和刷新源代码的权限。你必须下载一份商业版的

BitKeeper软件，具体做法如下：

- (1) 进入BitKeeper网站 (www.bitkeeper.com) 并单击Downloads链接。
- (2) 单击evaluation and download form链接。
- (3) 如实填写那份表格，选中Eval Key and Download Instructions选项。一定要给出你提出这个申请的理由以及打算如何使用MySQL源代码做一个简要的说明。
- (4) 如果申请获得了批准，你将收到一封来自BitKeeper的电子邮件，其内容是下载和安装商业版BitKeeper软件的详细步骤。

幸好，商业版BitKeeper软件在许多平台上都是具有GUI的。BitKeeper公司甚至还提供了一个能配合Visual Studio使用的客户端程序。上述工作告一段落之后，你应该马上打开BitKeeper客户端程序的自带文档，学习如何同步和刷新MySQL的源代码树。

在Windows平台上使用BitKeeper

在Windows平台上使用免费的BitKeeper客户端程序比较麻烦。你必须先下载和安装Cygwin。Cygwin是一个可以在Windows平台（NT、XP等）上编译和运行Linux程序的虚拟操作系统环境。安装好Cygwin之后，就可以按照我刚才介绍的步骤去下载免费的BitKeeper客户端程序，并创建MySQL源代码树副本了。以下是在Windows系统上下载、安装和使用免费版BitKeeper客户端程序的步骤：

(1) 从www.cygwin.com网站下载Cygwin的setup.exe文件。你应该在该网站的主页上看到一个Install or update now! 链接。

(2) 按照屏幕提示操作，不要改变默认的安装文件夹（相信我，这种方法更容易）。记得要把gcc、make（位于安装过程中的Select Packages屏幕上的Devel软件包里）以及所有必要的开发工具包全部安装到位。

(3) 从www.bitkeeper.com/Hosted.Downloading.html页面下载BitKeeper客户端程序。

(4) 把文件保存到c:\cygwin\username\文件夹里（username代表你登录子目录的名字）。

(5) 打开Cygwin命令窗口（安装程序会在你的Windows桌面上创建一个快捷方式）。

(6) 输入sh bk-client.shar命令。

(7) 输入cd bk_client-1.1命令改变工作目录。

(8) 用写字板打开c:\cygwin\home\username\bk_client-1.1文件夹里的makefile文件，把其中的\$(CC) \$(CFLAGS) -o -sfio -lz -sfio.c改为\$(CC) \$(CFLAGS) -o -sfio sfio.c-lz。注意：千万不要删除\$前面的制表符。

(9) 用make all命令编译BitKeeper客户端程序。注意：如果这个步骤失败了，请参见下面的“注意”去解决与getline函数有关的错误。

(10) 用PATH=\$PWD:\$PATH命令把你的路径改为当前文件夹（使用下面的路径字符串也可以）。

(11) 如果你想把MySQL源代码树放到另外一个地方而不是当前子目录，找到那个地方即可。

(12) 用sfio ball -r+ bk://mysql.bkbits.net/mysql-5.1 /home/username/mysql-5.1命令把源代码树复制到那个文件夹里。

现在，你的Windows系统里应该有MySQL源代码树的一份完整副本了。不过，这份副本在

Windows系统上用起来不是很方便，所以MySQL AB公司还特意为Windows程序员们准备了一些Visual Studio工程文件，但它们通常是在即将发布或刚刚发布MySQL源代码的GA版本（GA是Generally Available”的字头缩写，意思是“已经足够成熟和稳定”）时创建的，不一定就是你需要的东西。还好，只要你有一个功能完备的GNU开发环境，仍可以在Windows系统上编译MySQL源代码。根据我个人的经验，如果打算在Windows环境下研究MySQL源代码，选用GA版本会更容易一些。

下载链接:

- ❑ 作为产品投入使用的当前发布版本（也称为Generally Available版本或GA版本）。
- ❑ 正在开发的版本（如α版、β版等，请参见第1章中MySQL AB公司提供的版本的类别）。
- ❑ 软件的老版本。

在这个网页上还有许多MySQL辅助软件的下载链接，其中包括数据库连接器、管理工具等。

还可以使用源代码快照来下载源代码。那些快照通常是α版、开发版或GA版；β版本的下载链接一般都放在主页面上。如果你想看看某个新功能的最新进展，或者需要使用最新的稳定版本（稳定在这里的意思是“在测试中没有发现重大缺陷”），但不想或不需要使用源代码库时，源代码快照应该是个不错的选择。

如果你想试试本书后面内容里的例子，应该下载MySQL 5.1.7或更高版本。我将在下一节向大家介绍安装MySQL的步骤。需要提醒大家的是，那个网站上有无数的二进制代码和源代码下载链接，但它们适用于不同的平台，在下载时一定要根据自己的平台把源代码和二进制可执行文件都下载回来（两个下载文件），千万不要弄错了。在这本书里，我将使用的例子取材于Red Hat Linux Fedora Core 5和Microsoft Windows XP Professional平台。

提示 如果你使用的是Windows，一定要把所有的二进制文件或代码下载齐全。不要满足于最基本的软件包，它们可能缺少我们将在下一节提到的某些文件夹。

对OS/2的支持

在我撰写本书的时候，人们正在讨论要不要把对OS/2的支持从MySQL 5.1版本里删掉。MySQL AB公司今后很可能不再继续支持OS/2。但就算如此也用不着担心，在Planet MySQL博客网站上（www.planetmysql.org）有许多帖子，告诉人们从哪里可以找到支持OS/2的MySQL源代码变体。

注解 如果没有特别指出，本书的例子都来自适用于Linux平台的MySQL源代码发行版本（下载文件名是mysql-5.1.7-beta.tar.gz）。MySQL的绝大多数源代码既适用于Linux平台，又适用于Windows平台；而我会在必要时指出它们的差异。总的来说，适用于Windows平台的MySQL源代码有一个稍微不同的vio实现。

3.2 MySQL 源代码

下载完MySQL的源代码之后，还需要把那些文件解压缩到系统中的某个文件夹里。如果你愿意，直接在下载目录里对它们进行解压缩也可以。这么做的时候，你会看到大量的文件夹和源文件。最重要的文件夹是/sql文件夹，它里面的文件包含着MySQL服务器的大部分源代码。表3-1列出了与本书内容关系最密切的文件夹及其内容。

表3-1 MySQL的源代码文件夹

文件夹	内 容
/BUILD	编译配置并为所有被支持的平台制作文件。编译和链接MySQL源代码需要使用这个文件夹
/client	MySQL命令行客户端工具
/dbug	调试工具（详见第5章）
/Docs	随机文档。Linux用户还需要执行support文件夹里的generate-text-files.pl文件来生成那些文档。Windows用户直接打开manual.chm文件即可
/include	基本的系统包括文件和首部
/libmysql	用来创建嵌入式系统的MySQL客户端程序API，它是用C语言写的。（详见第6章）
/libmysqld	MySQL服务器的核心级API文件。也用来开发嵌入式系统。（详见第6章）
/mysql-test	MySQL系统的测试工具箱（详见第4章）
/mysys	核心级操作系统API的大部分封装函数和各种辅助函数
/regex	一个用来处理正则表达式的库。查询优化器和查询执行引擎在分析各种条件表达式时会用到它们
/scripts	一组基于shell脚本的工具
/sql	主系统代码。这个文件夹是我们学习和研究MySQL源代码的出发点
/sql-bench	一组性能测试工具
/SSL	一组SSL（Secure Socket Layer，安全套接层）工具和定义
/storage	MySQL插件式存储引擎的源代码就位于这个文件夹里。它还收录着存储引擎的示例代码（详见第7章）
/strings	各种字符串处理函数。在一个MySQL系统里，与字符串有关的所有处理都要使用这个函数来完成
/support-files	各种辅助文件，其中包括许多使用不同的选项编译MySQL源代码的配置文件
/tests	一组测试程序和测试文件
/vio	网络层和套接层的代码
/zlib	数据压缩工具

建议大家现在就去查看一下这些文件夹，熟悉有关文件的存放位置。你们将在这些文件夹里发现很多制作文件和各种各样的Perl脚本。总的来说，MySQL的源代码是按照源代码的功能而不是子系统来组织的。有些子系统（比如存储引擎）的源代码统一存放在某个子目录下，但绝大多数子系统的源代码散布在整个文件夹结构的各处。在剖析MySQL各子系统的源代码时，我将列出相关的源代码文件及其存放位置。

3.2.1 预备知识

要想了解MySQL系统的工作和控制流程，最好的办法是沿着一个典型的查询命令在MySQL里被处理的过程去分析相关的源代码。本书的第2章已经概括地介绍了MySQL的各个子系统。现在，我将按照同样的子系统流程描述一个典型的查询命令是如何被执行的。下面是将使用的样板SQL语句：

```
SELECT lname, fname, DOB FROM Employees WHERE Employees.department = 'EGR';
```

这个查询将把工程部里每一个人的姓名和出生日期查出来。虽然谈不上什么趣味性，但这个查询还是很有用的——它可以把MySQL系统里几乎所有的子系统展示出来。我们就从这个查询进入

MySQL服务器的地方开始讲起吧。

图3-1给出了这个样板查询穿过MySQL源代码的路径。图中的代码行是为了方便大家识别，从本书第2章所介绍的各个子系统特意抽取出来的main()函数不属于任何一个子系统，它负责对服务器进行初始化和创建连接监听器。main()函数的源代码在/sp1/mysql.cc文件里。

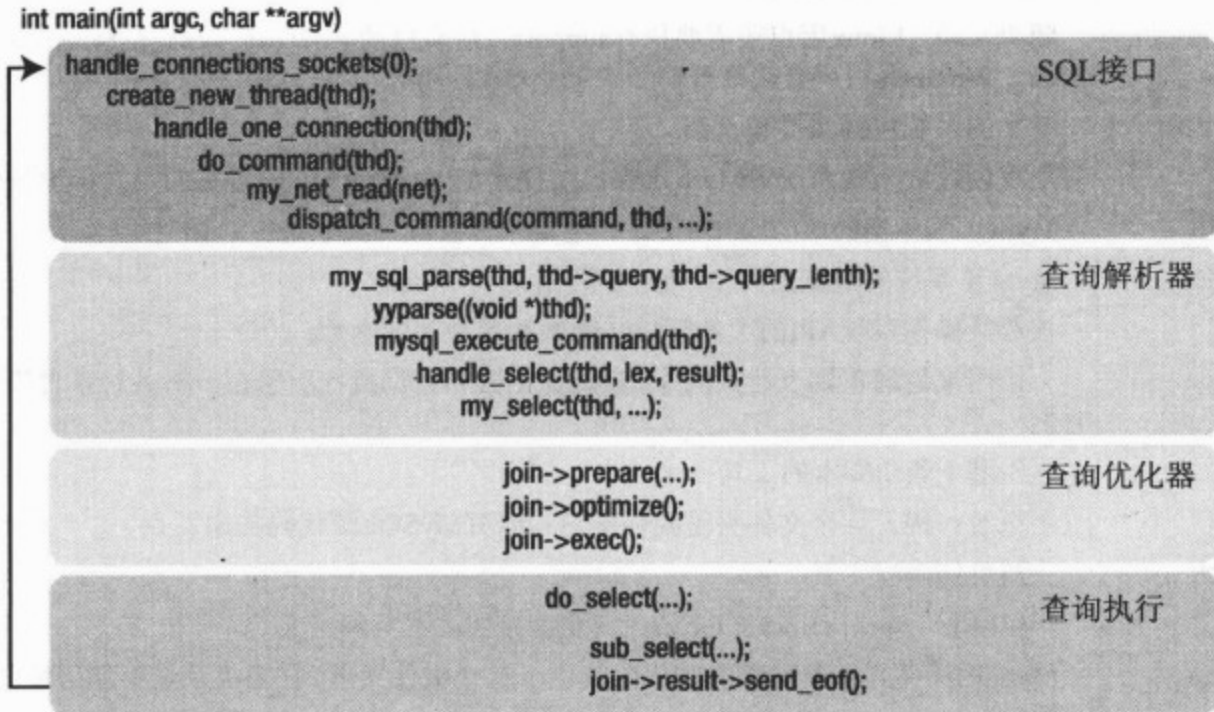


图3-1 查询处理流程概览

查询的处理流程从SQL接口子系统开始。（与绝大多数MySQL子系统一样，SQL接口子系统里的函数散布在一组松散关联的源代码文件里。）在接下来的内容里，我将告诉读者各有关的方法来自哪一个文件。handle_connections_socket()方法（来自/sql/mysql.cc文件）负责实现监听器循环，为监听到的每一个连接创建一个线程。线程被创建出来之后，控制权转到了handle_one_connection()函数。handle_one_connection()函数负责识别查询命令，然后把控制权转给do_command开关（来自/sql/sql_parse.cc文件）。do_command开关调用某个网络读取函数从连接读入查询命令，然后通过dispatch_command()函数（来自/sql/sql_parse.cc文件）把查询命令传递到解析器。

现在，查询命令进入了查询解析器子系统，它将在接受分析并被传递到优化器的正确部分。查询解析器是用Lex和YACC建立的。Lex用来识别各种记号、常数和语法。YACC用来生成与MySQL源代码交互的代码。查询解析器将把SQL命令分解成一系列基本元素、存入一个内部查询结构，再把查询命令传递给一个名为mysql_execute_command()（但这个名字有点儿名不符实）的命令处理器。这个方法将把查询命令传递给一个适当的子函数——my_select()函数。这些方法都来自/sql/sql_parse.cc文件。这部分代码且来输入“选取-投影-联结”查询优化器的“选取-投影”部分。

提示 “投影”是一个关系数据库术语，描述了把结果集限制到那些定义在SQL命令的列清单里的列，比如说，SQL命令SELECT lname, fname FROM employee将只把employee表里的fname和lname列“投影”到结果集里。

现在进入了查询优化器子系统。在join->prepare()和join->optimize()函数对查询的执行过程进

行优化之后，`join->optimize()`函数将开始实际执行这个查询，即控制权将被转给底层的`do_select()`函数以具体完成“选取”和“投影”操作。最后，`sub_selcet()`函数将调用某个存储引擎去读取并处理元组，然后把结果返回给客户。这些方法都来自`/sql/sql_select.cc`文件。在查询结果被写到网络之后，控制权将返回到`handle_connections_sockets`循环（来自`/sql/mysqld.cc`文件）。

提示 类，结构，类，结构——到处都是类和结构！在研究MySQL源代码的时候，一定要记住这一点。发生在MySQL服务器里的每一个操作最少也会有一个用来管理数据或控制执行流程的类或结构。正如你将在本章稍后的3.2.9节里看到的那样，是否熟悉常用的MySQL类或结构是能否正确理解MySQL源代码的关键。

看到这里，你也许会认为MySQL的源代码并不像别人说的那么难以理解，这很大程度上是因为我刚才选用的例子只是一个非常简单的SELECT语句而已，很快你就会看到它会变得相当复杂。在看过这个流程并知道了这个流程涉及的主要子系统和函数之后，你应该打开源代码并查阅那些函数。可以从`/sql/mysqld.cc`文件（Windows用户请查阅`/sql/mysqld.cpp`文件）开始你的探索。

提示 对于源文件，Windows源代码经常采用不同的扩展名。大多数情况下，你只需将`.cc`替换为`.cpp`就能得到对应的Windows源代码文件。在这一规则不适用的情况下，我会指出Linux和Windows文件之间的所有不同之处。

觉得上面那些介绍太简短？别着急。在接下来的几小节里，我将放慢节奏并告诉大家更多关于MySQL源代码的细节。我将在每个小节的末尾把有关的函数和源代码文件列在一个表格里。系好安全带，我们要出发了！

在你将要看到的各代码清单里，我将略掉一些与这次旅行关系不大的代码段。这些代码段包括条件编译指令、辅助代码和一些不太重要的系统级调用。我将以…来代表省略的代码段。我保留了许多原始的注释，相信它们可以帮助你看懂源代码并激起你开发一流数据库系统的雄心。最后，为了让大家更容易抓住重点，代码清单里的重要代码显示为黑体字。

3.2.2 main()函数

`main()`函数是MySQL服务器开始执行的地方，是服务器的可执行代码被加载到内存里以后执行的第一个函数。这个函数里的几百行代码是用来完成启动MySQL服务器和进行系统级初始化工作的。代码清单3-1给出了`main()`函数经过高度浓缩的框架，重要的部分都突出显示为黑体字。

代码清单3-1 main()函数

```
int main(int argc, char **argv)
{
    ...

    if (init_common_variables(MYSQL_CONFIG_NAME,
                             argc, argv, load_default_groups))
```

```

...

if (init_server_components())

...
/*
  Initialize my_str_malloc() and my_str_free()
*/
my_str_malloc= &my_str_malloc_mysql;
my_str_free= &my_str_free_mysql;

...

if (acl_init(opt_noacl) ||
    my_tz_init((THD *)0, default_tz_name, opt_bootstrap))

...

create_shutdown_thread();
create_maintenance_thread();

...

handle_connections_sockets(0);

...

(void) pthread_mutex_lock(&LOCK_thread_count);

...

(void) pthread_mutex_unlock(&LOCK_thread_count);

...
}

```

进入main()函数之后,我想介绍给大家的第一个子函数是init_common_variables()。这个函数使用命令行参数来控制MySQL服务器将如何工作,服务器就是在这里解读命令行参数,并根据那些参数而启动为各种模式的。这个函数负责设置系统变量和把服务器启动为指定的模式。init_server_components()函数负责对MySQL各子系统的一系列数据库日志进行初始化,这些日志记载着在MySQL系统内发生的各种事件、语句执行情况等。

我想特别提醒大家注意两个非常重要的my_库函数: my_str_malloc()和my_str_free。这两个函数指针是在服务器启动代码时(差不多刚刚开始执行)被设置的。这两个MySQL函数增强了出错处理方面的功能,安全性更有保障,所以你应该始终使用它们来代替C/C++语言里的malloc()函数。acl_init()函数的任务是启动MySQL的身份验证和访问控制子系统,这个子系统关系着整个系统的安全,所以才会这么早就出现在MySQL服务器的启动代码里。

接下来的两个函数将创建两个非常重要的辅助线程。create_shutdown_thread()函数创建的线程负责在收到有关信号时关闭MySQL，create_maintenance_thread()函数创建的线程负责处理各种服务器范围内的维护功能。我在下面标题为“进程与线程”的文本框里对线程做了更详细的讨论。

当启动代码执行到这个地方的时候，系统已经为接收来自客户端的连接做好了准备工作。接下来，handle_connections_sockets(0)函数实现了一个监听器，它将循环等待来自客户端的连接。我将在稍后的内容里对这个函数做进一步讨论。

我在这段代码里想为大家指出的最后一件事是在多线程环境里用于互斥访问的“关键节保护”(critical section protection)代码的例子。所谓“关键节”是一个必须作为一个整体来执行且每次只能被一个线程访问的代码块。在多线程环境里，对共享的内存变量进行写操作的代码块通常都属于关键节，这种写操作必须在另一个线程试图读取那块内存之前完成。MySQL AB公司把为MySQL系统创建的通用并发保护机制称为互斥法(mutex，是mutually exclusive的缩写，意思是“相互排斥”)。如果在你的代码里发现某个区域需要在并发执行期间受到保护，可以使用下面两个函数来保护该区域里的代码。

你应该调用的第一个函数是pthread_mutex_lock([resource reference])。这个函数将在代码中的这个位置设置一把锁。在代码调用解锁函数pthread_mutex_unlock([resource reference])之前，这把锁将不允许另一个线程去访问受它保护的内存区域。在main()函数里，这两个互斥法调用锁定的是全局性线程计数器变量。

这很可能是你第一次深入到MySQL源代码的内部。感觉如何？想知道更多的东西吗？就请大家继续看下去——这只是刚开个头而已。事实上，你还没看到示例查询将从什么地方进入MySQL系统；下一节就会看到了。

进程与线程

进程(process)和线程(thread)往往被人们混为一谈，这是不正确的。严格地讲，进程是一组有自己的内存和执行路径的计算机指令；线程虽然也是一组计算机指令，但线程都执行在一个宿主的执行路径里，没有自己的内存。(有些人把“线程”称为“轻量级进程”，这应该说是一种比较形象的比喻，但把它们称为轻量级进程丝毫无助于区分这两个概念。)它们确实可以保存状态(在MySQL里，这是通过THD类实现的)。简单地说，在谈论一个支持进程的大型系统时，我的意思是这个系统允许该系统的各有关部分作为一个彼此独立的进程来执行，每个进程都有自己的内存；在谈论一个支持线程的大型系统时，我的意思是这个系统允许该系统的各有关部分与其他部分并发执行，这些线程一起共享着同一块内存区域。

绝大多数现代数据库系统使用进程模型来管理并发连接和辅助函数。MySQL使用的是多线程模型。与使用进程相比，使用线程的优点主要有以下几点。首先，线程更容易创建和管理(没有内存分配/合并方面的开销)。其次，线程的切换非常迅速，因为无需切换任何上下文。不过，线程也有一个非常严重的先天不足。万一某个线程在执行期间出轨了(“出轨”在这里的意思是指发生了难以预料的行为，对线程而言，“出轨”通常是指奇怪和有害的事件)，那么，如果麻烦很严重的话，整个系统都有可能受到影响。让人感到欣慰的是，MySQL AB公司和全世界的程序员一直在努力让MySQL的线程子系统更加健壮和可靠。如果你也想为MySQL添砖加瓦，就必须注意线程的保护问题；这一点怎么强调也不过分。

3.2.3 处理连接和创建线程

在上一节中，你已经看到了MySQL系统如何启动，以及控制权如何转到等待用户连接的监听器循环上。连接是由客户端（程序）创建的，客户端软件将把它们拆分并封装为一组数据包，然后把那些数据包放到网络上。服务器的网络子系统将从网络通信线路上捕获那些数据包，并在服务器上把它们重新组合为数据。[通信数据包的完整描述可以在MySQL Internals Manual (MySQL内部工作原理手册)上查到。]图3-2描绘了这一过程。我将在下一章对MySQL中的网络通信函数（方法）做更详细的介绍，还为大家准备了几个利用这些函数把查询结果返回给客户端的编程示例。

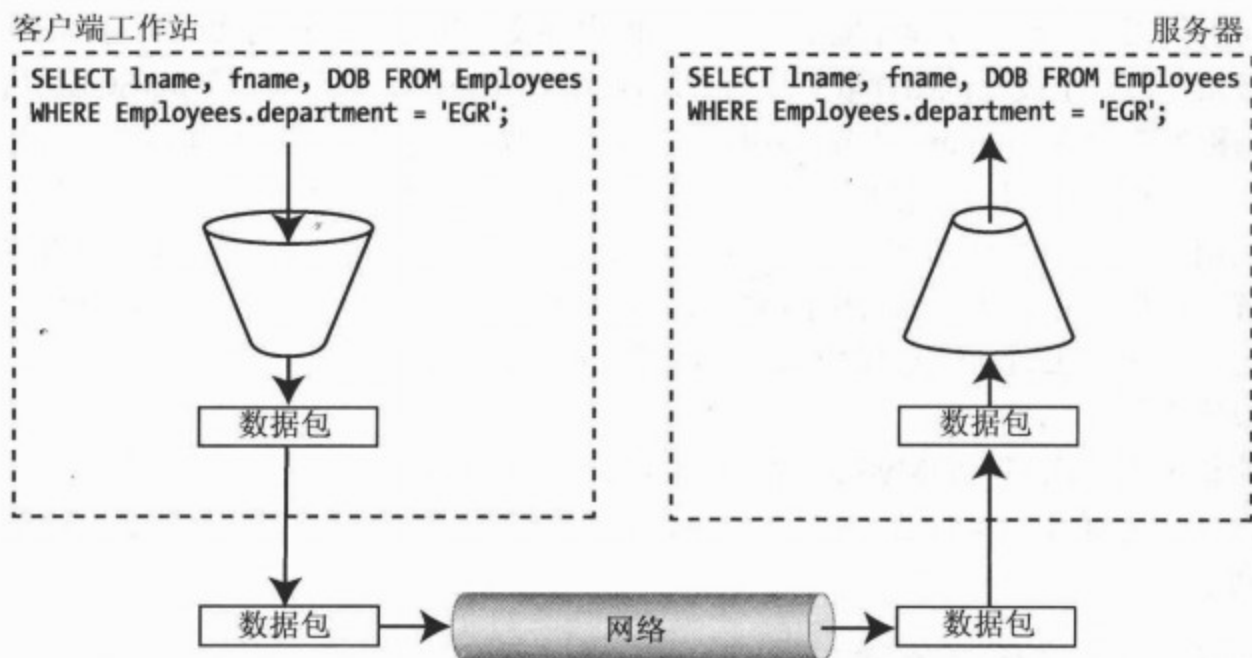


图3-2 从客户端到服务器的网络通信

在MySQL系统里，处理连接和创建线程的工作由SQL接口子系统负责。换句话说，数据包（包含着查询命令）已经到达了服务器，并通过handle_connections_sockets()函数被探测到。这个函数其实是一个无限循环，该循环的退出条件是变量abort_loop被设置为TRUE。负责管理连接和线程的MySQL源代码主要来自表3-2列出的几个文件。

表3-2 连接和线程管理

源代码文件	说 明
/sql/net_serv.cc	包含着所有的网络通信函数。如果你想知道怎样才能通过网络与MySQL客户端或服务 器进行通信，这个文件将提供许多信息
/include/mysql_com.h	绝大多数与网络通信有关的数据结构都是在这个文件里定义的
/sql/sql_parse.cc	除了词法解析器本身，对查询进行预处理和分析的大部分函数都来自这个文件
/sql/mysqld.cc	除main()和一些用来启动MySQL服务器的函数以外，这个文件还包含着用来创建线程 的方法

代码清单3-2是连接处理代码的浓缩版本。在探测到一个连接的时候（我省略了这部分代码，因为它们与正在讨论的问题关系不大），该函数调用create_new_thread()函数创建一个新的线程。最主要的MySQL数据结构当中的第一个结构就是在这个函数里被创建出来的。THD类负责保存和管理与线程

有关的所有信息。虽然THD类不是在一个独享的内存空间里被分配给线程的，但系统可以通过它对线程的执行情况进行全面的控制。我过一会儿再向大家介绍THD类的部分细节。

代码清单3-2 handle_connections_sockets()函数

```
pthread_handler_t handle_connections_sockets(void *arg __attribute__((unused)))
{
    ...

    DEBUG_PRINT("general",("Waiting for connections."));

    ...

    while (!abort_loop)
    {
        ...

        /*
         ** Don't allow too many connections
         */

        if (!(thd= new THD))

        ...

        if (sock == unix_sock)
            thd->security_ctx->host=(char*) my_localhost;

        ...

        create_new_thread(thd);
    }

    ...
}
```

现在，客户端已经与服务器连接上了。接下来会发生些什么？一起去看看create_new_thread()函数里会发生些什么。代码清单3-3是create_new_thread()函数的浓缩版本，你首先会看到一个用来锁定线程计数器的互斥法调用。正如你在main()函数里看到的那样，这是避免其他线程竞争对该变量进行写操作所必需的。在新线程被创建出来之后，一个相应的解锁互斥法调用将释放被锁定的资源。

代码清单3-3 create_new_thread()函数

```
static void create_new_thread(THD *thd)
{
    ...
```

```

pthread_mutex_lock(&LOCK_thread_count);

...

if (cached_thread_count > wake_thread)
{
    start_cached_thread(thd);
}
else
{
    int error;
    thread_count++;
    thread_created++;
    threads.append(thd);
    if (thread_count-delayed_insert_threads > max_used_connections)
        max_used_connections=thread_count-delayed_insert_threads;
    DEBUG_PRINT("info",(("creating thread %d"), thd->thread_id));
    thd->connect_time = time(NULL);
    if ((error=pthread_create(&thd->real_id,&connection_attrib,
        handle_one_connection,
        (void*) thd)))
    {
        DEBUG_PRINT("error",
            ("Can't create thread to handle request (error %d)",
            error));
    }
}

...

(void) pthread_mutex_unlock(&LOCK_thread_count);
}
DEBUG_PRINT("info",("Thread created"));

...
}

```

在这个函数开始的地方发生了一件很有趣的事。请注意start_cached_thread()函数调用，它试图在连接池（connection pool）里寻找一个现有的线程来重复使用。这是为了节约时间——虽然创建一个线程要比创建一个进程要快，但还是需要花一点儿时间的。准备好要执行的线程是连接的一种缓存机制。这种把线程保存起来以便后续使用的机制称为连接池。

如果在连接池里没能找到一个可以重复使用的连接（线程），系统将调用pthread_create()函数创建一个新的连接。这里有几个需要大家注意的地方。请注意这个函数调用的第三个参数，这个看起来像是一个变量的参数，其实是一个函数的入口地址（一个函数指针）。pthread_create()函数需要使用这个函数指针把新线程与它在服务器内存中的执行入口地址关联在一起。

现在，用户发出的查询命令已经从客户端到达了服务器，服务器也为管理该查询的执行情况创建出了一个线程。接下来，控制权转到了handle_one_connection()函数。代码清单3-4是handle_one_connection()函数

的浓缩版本。我在这个浓缩版本里省略了一大段用来对THD类进行初始化的代码；如果你对那些代码感兴趣，建议你另外找个时间好好研究一下它们（位于/sql/mysqld.cc文件）。我在这里只介绍这个函数的基本工作情况。

代码清单3-4 handle_one_connection()函数

```
pthread_handler_t handle_one_connection(void *arg)
{
    THD *thd=(THD*) arg;

    ...

    while (!net->error && net->vio != 0 &&
           !(thd->killed == THD::KILL_CONNECTION))
    {
        net->no_send_error= 0;
        if (do_command(thd))
            break;
    }

    ...
}
```

浓缩后的handle_one_connection()函数里只剩下一个与这次探索有关系的函数：do_command()(thd)函数。它位于一个循环的内部，该循环的作用是对从网络通信代码读入的所有命令依次进行处理。这没有什么好奇怪的，因为有些用户喜欢在同一个命令行上连续输入多条SQL命令。正如你看到的那样，这是MySQL开始对查询命令进行具体处理的地方。根据读入的每一条命令，该函数将控制传递给从网络开始在查询中执行读操作的函数。

系统就是在这里从网络读入查询命令并放入THD类，以便在后续步骤里进行分析。这些事就发生在do_command()函数里。代码清单3-5是do_command()函数的浓缩版本。我保留了一些比较意思的注释和代码，你可以从中感受到MySQL源代码的健壮性。

代码清单3-5 do_command()函数

```
bool do_command(THD *thd)
{
    char *packet;
    uint old_timeout;
    ulong packet_length;
    NET *net;
    enum enum_server_command command;

    ...

    packet=0;

    ...
}
```

```

net_new_transaction(net);
if ((packet_length=my_net_read(net)) == packet_error)
{
    DEBUG_PRINT("info",("Got error %d reading command from socket %s",
        net->error,
        vio_description(net->vio)));
    ...
}
else
{
    packet=(char*) net->read_pos;
    command = (enum enum_server_command) (uchar) packet[0];
    if (command >= COM_END)
        command= COM_END;        // Wrong command
    ...
}
net->read_timeout=old_timeout;    // restore it
/*
    packet_length contains length of data, as it was stored in packet
    header. In case of malformed header, packet_length can be zero.
    If packet_length is not zero, my_net_read ensures that this number
    of bytes was actually read from network. Additionally my_net_read
    sets packet[packet_length]= 0 (thus if packet_length == 0,
    command == packet[0] == COM_SLEEP).
    In dispatch_command packet[packet_length] points beyond the end of packet.
*/
DEBUG_RETURN(dispatch_command(command,thd, packet+1, (uint) packet_length));
}

```

do_command()函数一上来先创建了一个packet缓冲区和一个NET结构。packet缓冲区其实是一个字符数组，用来存放从网络读入并被存入NET结构的查询命令字符串。然后创建了一个command结构，这个结构被用来将控制权转交给相应的解析器函数。从网络读取各有关数据包并将其存入NET结构的工作是由my_net_read函数负责具体完成的。数据包的长度也被保存在NET结构的packet_length变量里。你在这个函数里看到的最后一条语句是一个dispatch_command()调用，查询命令就是被这个调用送入MySQL服务器代码接受各种处理的。

好了，查询命令现在将去往某个地方。dispatch_command()函数的工作是把控制权转给最适合处理当前查询命令的那部分服务器代码。因为我们的示例SQL语句是一个普通的SELECT查询，所以系统已经通过把command变量设置为COM_QUERY的办法把该语句标识为一个查询。其他的命令类型用来标识语句、改变用户、生成统计报告和许多其他的服务器功能。在这一章里，我将只分析真正的查询命令(COM_QUERY)。代码清单3-6是dispatch_command()函数的浓缩版本。为简明起见，我省略了对应于其他命令类型的switch分支的具体代码（以及注释），但保留了绝大多数命令类型的case语句。花点儿时间去看看这个列表。你可以从绝大多数case分支的名字中窥见其用意。如果想分析其他类型的查询命令，

就应该在这个函数里根据你准备分析的查询命令的具体类型，去查看相应的case分支里的代码。我还保留了位于这个函数开头部分的一大段注释，这段注释对整个函数功能的用法进行了说明。花点儿时间去看看那些注释，本章稍后还会提到它们。

代码清单3-6 dispatch_command()函数

```

/* \
    Perform one connection-level (COM_XXXX) command.

SYNOPSIS
    dispatch_command()
    thd                connection handle
    command            type of command to perform
    packet             data for the command, packet is always null-terminated
    packet_length      length of packet + 1 (to show that data is
                        null-terminated) except for COM_SLEEP, where it
                        can be zero.

RETURN VALUE
    0  ok
    1  request of thread shutdown, i. e. if command is
        COM_QUIT/COM_SHUTDOWN
*/

bool dispatch_command(enum enum_server_command command, THD *thd,
                     char* packet, uint packet_length)
{
    ...
    switch (command) {
    case COM_INIT_DB:
    ...
    case COM_REGISTER_SLAVE:
    ...
    case COM_TABLE_DUMP:
    ...
    case COM_CHANGE_USER:
    ...
    case COM_STMT_EXECUTE:
    ...
    case COM_STMT_FETCH:
    ...
    case COM_STMT_SEND_LONG_DATA:
    ...
    case COM_STMT_PREPARE:
    ...
    case COM_STMT_CLOSE:
    ...
    case COM_STMT_RESET:
    ...
    ...

```



```

case COM_QUERY:
{
    if (alloc_query(thd, packet, packet_length))
        break;          // fatal error is set

    ...

    general_log_print(thd, command, "%s", thd->query);

    ...

    mysql_parse(thd, thd->query, thd->query_length);

    ...
}
case COM_FIELD_LIST:      // This isn't actually needed
...
case COM_QUIT:
...
case COM_BINLOG_DUMP:
...
case COM_REFRESH:
...
case COM_STATISTICS:
...
case COM_PING:
...
case COM_PROCESS_INFO:
...
case COM_PROCESS_KILL:
...
case COM_SET_OPTION:
...
case COM_DEBUG:
...
case COM_SLEEP:
...
case COM_DELAYED_INSERT:
...
case COM_END:
...
default:
...
}

```

进入COM_QUERY处理程序之后所发生的第一件事是调用alloc_query()函数把查询命令从packet数组复制到了thd->query成员变量。这样一来，线程现在就有了一份查询命令的副本了，这个副本将伴随着线程直到它执行结束。此外，有关代码还把这条查询命令写入了MySQL系统的操作日志，这些信息可以帮助我们调试系统故障和查询问题。代码清单3-6里的最后一个让人感兴趣的函数调用是

mysql_parse()函数调用。这个调用是SQL接口子系统与查询分析子系统的“官方”分界线。不过，正如你将看到的那样，这条分界线的语义含义大于语法含义。

3.2.4 解析查询

解析查询的工作终于开始了。MySQL服务器将从这里开始对查询命令进行解析和执行。解析器代码散布在好几个文件里（MySQL系统的大部分子系统都是这样）。在分析MySQL代码的时候，只要你记住了下面这句话，事情就不会太难：MySQL的源代码是按功能而不是按体系结构组织的。

你现在正在查看的函数是mysql_parse()函数（来自/sql/sql_parse.cc文件）。它的主要工作有3项：检查查询缓存里是否有以前执行过的某个查询与当前查询有同样的结果集，把控制权转到词法解析器，把查询命令传递到查询优化器。代码清单3-7是mysql_parse()函数的浓缩版本。

代码清单3-7 mysql_parse()函数

```
void mysql_parse(THD *thd, char *inBuf, uint length)
{
    ...

    if (query_cache_send_result_to_client(thd, inBuf, length) <= 0)
    {
        LEX *lex= thd->lex;

        ...

        if (!yyparse((void *)thd) && ! thd->is_fatal_error)
        {
            ...

            mysql_execute_command(thd);
            query_cache_end_of_result(thd);

            ...
        }
        ...
    }
}
```

值得注意的第一件事是对查询缓存的调用。查询缓存里存放着最近一段时间以来出现频率最高的查询命令和它们查询结果。如果能在查询缓存里找到一个与当前查询有着同样结果集的查询，后面的步骤就都可以省略了。不需要再对当前查询继续进行解析、优化和执行了。怎么样，够酷的吧？

因为我们现在是在分析MySQL的源代码，所以我们需要假设在查询缓存里不能找到示例查询。于是，mysql_parse()函数将创建一个新的LEX结构存放这条查询命令的内部表示。这个结构的内容是由MySQL里的Lex/YACC解析器填写的，代码清单3-8给出了有关的代码。

代码清单3-8 SELECT命令的Lex/YACC解析代码(节选)

```

select:
    select_init
    {
        LEX *lex= Lex;
        lex->sql_command= SQLCOM_SELECT;
    }
;
/* Need select_init2 for subselects. */
select_init:
    SELECT_SYM select_init2
    |
    '(' select_paren ')' union_opt;

select_paren:
    SELECT_SYM select_part2
    {
        LEX *lex= Lex;
        SELECT_LEX * sel= lex->current_select;
        if (sel->set_braces(1))
        {
            yyerror(ER(ER_SYNTAX_ERROR));
            YYABORT;
        }
        if (sel->linkage == UNION_TYPE &&
            !sel->master_unit()->first_select()->braces)
        {
            yyerror(ER(ER_SYNTAX_ERROR));
            YYABORT;
        }
        /* select in braces, can't contain global parameters */
        if (sel->master_unit()->fake_select_lex)
            sel->master_unit()->global_parameters=
                sel->master_unit()->fake_select_lex;
    }
    | '(' select_paren ')';

select_init2:
    select_part2
    {
        LEX *lex= Lex;
        SELECT_LEX * sel= lex->current_select;
        if (lex->current_select->set_braces(0))
        {
            yyerror(ER(ER_SYNTAX_ERROR));
            YYABORT;
        }
        if (sel->linkage == UNION_TYPE &&
            sel->master_unit()->first_select()->braces)

```



```

        {
            yyerror(ER(ER_SYNTAX_ERROR));
            YYABORT;
        }
    }
    union_clause
;

select_part2:
{
    LEX *lex= lex;
    SELECT_LEX *sel= lex->current_select;
    if (sel->linkage != UNION_TYPE)
        mysql_init_select(lex);
    lex->current_select->parsing_place= SELECT_LIST;
}
select_options select_item_list
{
    Select->parsing_place= NO_MATTER;
}
select_into select_lock_type;

select_into:
    opt_order_clause opt_limit_clause {}
    | into
    | select_from
    | into select_from
    | select_from into;

select_from:
    FROM join_table_list where_clause group_clause having_clause
    opt_order_clause opt_limit_clause procedure_clause
    | FROM DUAL_SYM where_clause opt_limit_clause
    /* oracle compatibility: oracle always requires FROM clause,
       and DUAL is system table without fields.
       Is "SELECT 1 FROM DUAL" any better than "SELECT 1" ?
       Hmmmm :) */
;

select_options:
    /* empty*/
    | select_option_list
    {
        if (Select->options & SELECT_DISTINCT && Select->options & SELECT_ALL)
        {
            my_error(ER_WRONG_USAGE, MYF(0), "ALL", "DISTINCT");
            YYABORT;
        }
    }
;

```

```

select_option_list:
    select_option_list select_option
    | select_option;

select_option:
    STRAIGHT_JOIN { Select->options|= SELECT_STRAIGHT_JOIN; }
    | HIGH_PRIORITY
    {
        if (check_simple_select())
            YYABORT;
        Lex->lock_option= TL_READ_HIGH_PRIORITY;
    }
    | DISTINCT { Select->options|= SELECT_DISTINCT; }
    | SQL_SMALL_RESULT { Select->options|= SELECT_SMALL_RESULT; }
    | SQL_BIG_RESULT { Select->options|= SELECT_BIG_RESULT; }
    | SQL_BUFFER_RESULT
    {
        if (check_simple_select())
            YYABORT;
        Select->options|= OPTION_BUFFER_RESULT;
    }
    | SQL_CALC_FOUND_ROWS
    {
        if (check_simple_select())
            YYABORT;
        Select->options|= OPTION_FOUND_ROWS;
    }
    | SQL_NO_CACHE_SYM { Lex->safe_to_cache_query=0; }
    | SQL_CACHE_SYM
    {
        Lex->select_lex.options|= OPTION_TO_QUERY_CACHE;
    }
    | ALL { Select->options|= SELECT_ALL; }
    ;

select_lock_type:
    /* empty */
    | FOR_SYM UPDATE_SYM
    {
        LEX *lex=Lex;
        lex->current_select->set_lock_for_tables(TL_WRITE);
        lex->safe_to_cache_query=0;
    }
    | LOCK_SYM IN_SYM SHARE_SYM MODE_SYM
    {
        LEX *lex=Lex;
        lex->current_select->
            set_lock_for_tables(TL_READ_WITH_SHARED_LOCKS);
        lex->safe_to_cache_query=0;
    }

```

```

;

select_item_list:
    select_item_list ',' select_item
| select_item
| '*'
{
    THD *thd= YYTHD;
    if (add_item_to_list(thd,
                        new Item_field(&thd->lex->current_select->
                                      context,
                                      NULL, NULL, "")))
        YYABORT;
    (thd->lex->current_select->with_wild)++;
};

select_item:
    remember_name select_item2 remember_end select_alias
{
    if (add_item_to_list(YYTHD, $2))
        YYABORT;
    if ($4.str)
    {
        $2->set_name($4.str, $4.length, system_charset_info);
        $2->is_autogenerated_name= FALSE;
    }
    else if (!$2->name) {
        char *str = $1;
        if (str[-1] == '`')
            str--;
        $2->set_name(str, (uint) ($3 - str), YYTHD->charset());
    }
};

```

以上代码节选自MySQL的Lex/YACC解析器定义文件，有关的YACC代码就是根据这些规则对SELECT记号进行识别和分析的。即使你不熟悉Lex或YACC，这段代码也不难看懂：只要留意代码里的关键字（或者叫记号）就可以找到相应的规则。这些关键字都以select:——关键字加冒号的形式单独出现在代码行的最左端，它们的作用是把控制权转交给解析器的相应部分。冒号后面的记号定义了应该按照什么顺序去分析查询命令里的关键字。以关键字select:为例，在冒号后面是select_init2关键字，它本身没有提供什么信息，所以我们需要在这段代码里找到select_init:关键字。这种安排使得Lex/YACC程序员可以像定义一条宏命令那样对Lex/YACC解析器的特定行为进行定义。另外，请注意select_init:关键字下面的那些花括号，解析器将根据这些花括号里定义规则把一个SQL命令字符串分解成一系列基本元素，并把它们存入LEX结构。常数型符号（比如“SELECT”）是在一个头文件（/sql/lex.h）里定义的；SELECT对应着解析器里的SELECT_SYM。请大家趁现在这个机会快速浏览一下代码清单3-8。如果以前没有研究过编译器的构造或文本解析技术，那可能需要多看几遍才能弄明白。

看不懂这些代码也没有关系，Lex/YACC代码对绝大多数程序员来说都是个挑战。代码清单3-8里

有几行以黑体字突出显示的关键代码，可以帮助大家看懂这一大段代码。下面，我将逐条地重点分析一下那几行代码。为方便大家查阅，我先把示例SELECT语句在这里重复一遍：

```
SELECT lname, fname, DOB FROM Employees WHERE Employees.department = 'EGR';
```

先去看看第一个关键字select:，请注意select_init代码块是如何把LEX结构的sql_command变量设置为SQLCOM_SELECT的。这很重要，因为查询处理路径的下一个函数需要根据这个变量从一长串switch语句中挑选一个分支，对这条查询命令做进一步处理。示例SELECT语句在字段清单里列出了3个字段。请找到相应的解析器代码（关键字select_item-list:）。请注意那个add_item_to_list()函数调用，解析器就是在这里分析出各个字段并把它们放到LEX结构里的。在这个调用的前几行，还可以看到用来识别字段清单里的*选项的解析器代码。现在，负责设置sql_command变量和识别各个字段的解析器代码都已经找到了。那么FROM子句又是由哪个部分负责分析呢？请大家找到以FROM join_table_list where_clause开头的代码语句（关键字select_from:），它们就是负责识别FROM和WHERE子句（还有其他子句）的解析器代码。负责具体处理这些子句的解析器代码没有包括在代码清单3-8里，但我想这应该不会影响你的思路。如果现在就想知道FROM子句里的表清单和WHERE子句里的表达式是如何填充到LEX结构的，只要打开/sql子目录里的sql_yacc.yy源代码文件就可以看到了。

注解 MySQL的某些Windows发行版本没有收录sql_yacc.yy文件。如果你使用的是Windows，并且在/sql子目录里没有找到这个文件，则需要下载适用于Linux平台的sql_yacc.yy源代码文件（文件名不变），并把它放到/sql子目录里去。

MySQL系统的解析器代码的确有点儿让人望而生畏，希望上面那些内容能够帮助你减轻这种恐惧。在本书的第8章里，我还会回到MySQL系统的这个部分，介绍如何给MySQL的SQL词法解析器增加新命令。表3-3列出了与MySQL解析器相关的源代码文件。

表3-3 MySQL解析器

源代码文件	说 明
/sql/lex.h	由MySQL解析器支持的所有关键字和记号构成的符号表
/sql/lex_symbol.h	符号表的类型定义
/sql/lex_hash.h	把符号映射到解析器里的各有关函数
/sql/sql_lex.h	LEX结构的定义
/sql/sql_lex.cc	Lex类的定义
/sql/sql_yacc.yy	Lex/YACC解析器代码
/sql/sql_parse.cc	除了词法解析器本身，对查询进行预处理和分析的大部分函数都来自这个文件

注意 千万不要直接编辑sql_yacc.cc、sql_yacc.h和lex_hash.h文件。这几个文件是由其他工具生成的。详见本书第8章的有关内容。

3.2.5 优化查询的准备工作

虽然MySQL的随机文档没有明确地为解析器和优化器划定一个分界线（有些说法彼此矛盾），但根据优化器的定义，与优化工作有关的switch分支和用来调控优化过程的源代码部分都应该是优化器的组成部分。为避免引起混乱，我将把下面将要介绍的那些函数称为“优化器的预备阶段”。

这些预优化函数中的第一个是mysql_execute_command()函数（来自/sql/sql_parse.cc文件）。这个名字很容易让人以为它真的是在执行查询命令，但实际情况却并非如此。在真正开始优化一个查询之前，还有一系列的设置和准备工作需要完成，其中大部分将由这个函数负责完成。为了让查询的优化和执行工作能够顺利进行，还需要复制LEX结构和设置一些变量。你可以在代码清单3-9给出的这个函数的浓缩版本里找到一些这样的操作。

代码清单3-9 mysql_execute_command()函数

```
bool mysql_execute_command(THD *thd)
{
    bool res= FALSE;
    int result= 0;
    LEX *lex= thd->lex;
    /* first SELECT_LEX (have special meaning for many of non-SELECT commands) */
    SELECT_LEX *select_lex= &lex->select_lex;
    /* first table of first SELECT_LEX */
    TABLE_LIST *first_table= (TABLE_LIST*) select_lex->table_list.first;
    /* list of all tables in query */
    TABLE_LIST *all_tables;
    /* most outer SELECT_LEX_UNIT of query */
    SELECT_LEX_UNIT *unit= &lex->unit;
    /* Saved variable value */
    DEBUG_ENTER("mysql_execute_command");
    thd->net.no_send_error= 0;

    ...

    switch (lex->sql_command) {
        case SQLCOM_SELECT:
        {
            ...

            select_result *result=lex->result;

            ...

            res= check_access(thd,
                lex->exchange ? SELECT_ACL | FILE_ACL : SELECT_ACL,
                any_db, 0, 0, 0, 0);

            ...
        }
    }
}
```

```

    if (!(res= open_and_lock_tables(thd, all_tables)))
    {
        if (lex->describe)
        {
            /*
             We always use select_send for EXPLAIN, even if it's an EXPLAIN
             for SELECT ... INTO OUTFILE: a user application should be able
             to prepend EXPLAIN to any query and receive output for it,
             even if the query itself redirects the output.
            */

            ...

            query_cache_store_query(thd, all_tables);
            res= handle_select(thd, lex, result, 0);

            ...
        }
    }

```

这个函数里有许多让人感兴趣的东西，其中最引人注目的是一个按照SQLCOM关键字进行分支的switch语句。具体到示例查询命令，你已经在前一小节看到了解析器是如何把lex-> sql_command成员变量设置为SQLCOM_SELECT的，而我在代码清单3-9里也为你保留了SQLCOM_SELECT分支的代码（经过浓缩）。这个函数非常大，它的源代码如果打印出来会有好几十页。这并不奇怪，因为它相当于查询处理工作的中央控制台，用户有可能会用到的每一种命令都有相应的case分支。

我们一起去看看这个函数将干些什么。首先，请注意select_result *result=lex->result语句。这条语句将创建一个result类来容纳将被传输给客户的查询结果。再往下，你将看到一个check_table_access()函数。这个函数负责检查当前查询将要用到的资源的访问控制表（access control list, ACL）。如果权限允许，这个函数将调用open_and_lock_tables()函数为当前查询命令打开并锁定有关的表。你还可以在代码清单3-9的后半部分看到一些与DESCRIPTION (EXPLAIN) 命令有关的注释。

注解 有一次，我需要从MySQL源代码里把所有的EXPLAIN调用找出来并加以修改。我找遍了所有地方，最后才在解析器里找到。我在MySQL解析器的Lex/YACC代码中间看到了这样一条注释：DESCRIBE命令与早期的Oracle不兼容，正确的做法是使用EXPLAIN命令。注释是有用的，但你必须找得到才行。

下一个函数调用是查询缓存调用。query_cache_store_query()函数将把SQL命令保存到查询缓存里。正如稍后将会看到的那样，检索出来的查询结果也被保存到查询缓存里。最后，这个函数将调用另外一个名为handle_select()的函数，你或许会问：难道我们不是正在进行处理吗？

handle_select()函数是另一个名为mysql_select()的函数的打包器（wrapper，也叫封装器）。代码清单3-10给出了handle_select()函数的完整代码。这份代码清单的开头部分是一个select_lex->next_select()操作，负责检查SQL语句里有没有用来把多组SELECT结果合并为一个结果集的UNION命令。在此之后，这段代码将顺序调用下一个函数，也就是mysql_select()。到了这里，你已经站在了MySQL查询优化器子系统的门口了。表3-4列出了与查询优化器相关的源代码文件。

注解 这部分代码或许是子系统之间界限不清的最大受害者了。这些代码本身都很有条理，但子系统之间的分界线却非常模糊。

代码清单3-10 handle_select()函数

```
bool handle_select(THD *thd, LEX *lex, select_result *result,
                  ulong setup_tables_done_option)
{
    bool res;
    register SELECT_LEX *select_lex = &lex->select_lex;
    DEBUG_ENTER("handle_select");
    if (select_lex->next_select())
        res= mysql_union(thd, lex, result, &lex->unit, setup_tables_done_option);
    else
    {
        SELECT_LEX_UNIT *unit= &lex->unit;
        unit->set_limit(unit->global_parameters);
        /*
         * 'options' of mysql_select will be set in JOIN, as far as JOIN for
         * every PS/SP execution new, we will not need to reset this flag if
         * setup_tables_done_option changed for next execution
         */
        res= mysql_select(thd, &select_lex->ref_pointer_array,
                          (TABLE_LIST*) select_lex->table_list.first,
                          select_lex->with_wild, select_lex->item_list,
                          select_lex->where,
                          select_lex->order_list.elements +
                          select_lex->group_list.elements,
                          (ORDER*) select_lex->order_list.first,
                          (ORDER*) select_lex->group_list.first,
                          select_lex->having,
                          (ORDER*) lex->proc_list.first,
                          select_lex->options | thd->options |
                              setup_tables_done_option,
                          result, unit, select_lex);
    }
    DEBUG_PRINT("info", ("res: %d report_error: %d", res,
                        thd->net.report_error));
    res|= thd->net.report_error;
    if (unlikely(res))
    {
        /* If we had another error reported earlier then this will be ignored */
        result->send_error(ER_UNKNOWN_ERROR, ER(ER_UNKNOWN_ERROR));
        result->abort();
    }
    DEBUG_RETURN(res);
}
```

表3-4 查询优化器

源代码文件	说 明
/sql/sql_parse.cc	解析器的绝大部分代码都包含在这个文件里
/sql/sql_select.cc	包含着一些优化函数和一些数据检索算法的具体实现
/sql/sql_parse.cc	除了词法解析器本身，对查询进行预处理和分析的大部分函数都来自这个文件

3.2.6 优化查询

马上就要见到真正的优化器了！不过，如果用这个名字去寻找源代码文件或类的话，你将一无所获。JOIN类倒是包含着一个名为optimize()的方法，但并不是我们想要找的东西。优化器是那些为查询命令寻找一条最短执行路径的流程控制函数和功能子函数的总称。我们要回答的问题是：那些优化算法、查询路径以及编译查询功能都是如何实现的？我在本书第2章里讨论MySQL的体系结构时曾经说过，MySQL的查询优化器是一种非传统的混合式优化器，是一些公认的优化规则与基于开销的路径选择技术相结合的产物。那些优化规则就是在这里开始发挥作用的。

人们普遍接受的优化规则有很多，对WHERE子句表达式里的参数进行标准化处理，是最好的优化实践之一。在我们的示例查询命令里，WHERE子句只包含一个表达式Employee.department = 'EGR'。这个表达式完全可以写成“'EGR' = Employee.department”（将返回同样的结果）。这也正是那些传统的基于开销的优化器能生成多种执行计划的原因——为同一个表达式的每一种等价变体分别生成一个计划。MySQL的优化器集成了许许多多的优化规则，下面只是其中的几个：

- 常数替代——尽可能地把复杂的关系运算简化为基本操作，能简化为常数最好。比如说，“a=b='c'”可以被简化为“a='c'”；这步优化消除了不必要的内部等式，减少了总的计算次数。再比如说，SQL命令SELECT * FROM table1 WHERE column1 = 12 AND NOT (column3 = 17 OR column1 = column2)可以被简化为SELECT * FROM table1 WHERE column1 = 12 AND column3 <> 17 AND column2 <> 12。
- 无效条件消除——去掉恒真条件。比如说，表达式“a=b AND 1=1”里的“AND 1=1”就是一个恒真条件，去掉不会影响查询结果。对恒假条件也可以做类似处理，即把那些对查询结果没有影响的“假”表达式去掉。比如说，SQL语句SELECT * FROM table1 WHERE column1 = 12 AND column2 = 13 AND column1 < column2可以被简化为SELECT * FROM table1 WHERE column1 = 12 AND column2 = 13。
- 范围查询——把IN子句转换为布尔表达式。比如说，IN (1, 2, 3)可以转换为a = 1 or a = 2 or a = 3。这种转换有助于简化表达式的求值运算。比如说，SQL语句SELECT * FROM table1 WHERE column1 = 12 OR column1 = 17 OR column1 = 21可以被简化为SELECT * FROM table1 WHERE column1 in (12, 17, 21)。

希望这几个例子能够让读者对世界上最成功的非传统查询优化器之一的内部机制有所了解。简单地说，它在查询量非常惊人的情况下也工作得非常好。

不过，在查看mysql_select()函数的代码时，你只能看到它识别联结操作和调用join->optimize()函数，看不到什么优化功能。那些优化规则都藏在什么地方呢？它们都在JOIN类里！JOIN类非常复杂，若是把对JOIN类的分析写成一本书的话，那本书的篇幅恐怕要比你手里的这本书还要长。这么说吧，对优化器进行剖析是一件复杂而又艰难的差事。还好，需要我们对MySQL系统进行如此深入剖析的情况非常少，但MySQL AB公司欢迎大家这么做！我衷心希望我在本小节对optimize()函数的概括性介

绍，能够成为你们钻研MySQL优化器的出发点。

要说与JOIN类有什么关系的话，你确实可以在optimize()函数里看到一行用来定义本地JOIN类对象的代码语句JOIN *join，接下来还可以看到这个函数去检查select_lex类是否包含JOIN类的对象。为什么要这么做？为了节省时间和资源：如果你在执行一个UNION里的另一条SELECT语句，或者如果你是在重复使用来自连接池的某个“老”线程的话，二次使用的select_lex类就应该已经被这部分代码处理过一次了，不需要再创建一个新的JOIN类。如果select_lex类里没有JOIN类，会由创建语句join = new JOIN()来创建一个新的。最后，你将看到这个函数调用了join->optimizer()方法。

现在，我们再次遇到了分界线比较模糊的情况，它这次发生在mysql_select()函数的中部。该函数里的下一个主要调用是join->exec()方法。代码清单3-11是mysql_select()函数的浓缩版本。表3-5列出了与查询优化有关的源代码文件。

代码清单3-11 mysql_select()函数

```
bool mysql_select(THD *thd, Item ***rref_pointer_array,
    TABLE_LIST *tables, uint wild_num, List<Item> &fields,
    COND *conds, uint og_num, ORDER *order, ORDER *group,
    Item *having, ORDER *proc_param, ulong select_options,
    select_result *result, SELECT_LEX_UNIT *unit,
    SELECT_LEX *select_lex)
{
    bool err;
    bool free_join= 1;
    DBUG_ENTER("mysql_select");

    select_lex->context.resolve_in_select_list= TRUE;
    JOIN *join;
    if (select_lex->join != 0)
    {
        join= select_lex->join;

        ...

        join->select_options= select_options;
    }
    else
    {
        if (!(join= new JOIN(thd, fields, select_options, result)))
            DBUG_RETURN(TRUE);

        ...
    }

    if ((err= join->optimize()))
    {
        goto err;
    }
}
```



```

...

join->exec();

...
}

```

表3-5 查询优化

源代码文件	说 明
/sql/sql_select.h	实现SELECT命令的各种数据检索函数里用到的各种结构的定义
/sql/sql_select.cc	包含着一些优化函数和一些数据检索算法的具体实现

3.2.7 执行查询

类似于优化器的情况，MySQL的查询执行子系统也使用了一组大家公认的优化规则来执行查询命令。比如说，在检测到ORDER BY和DISTINCT等子句的时候，查询执行子系统就将这些操作的控制权发送给快速排序方法和元组筛选方法。

这些活动中的绝大多数都发生在JOIN类的方法里。代码清单3-12是join::exec()方法的浓缩版本。请注意名为do_select()的函数，尤其是它的调用参数。看到“field list”字样了吗？这是否意味着我们就要开始从数据库读取数据了呢？是的，do_select()函数实际上是一个高级打包器，它就是用来做这件事的。

代码清单3-12 join::exec()函数

```

void JOIN::exec()
{
    List<Item> *columns_list= &fields_list;
    int      tmp_error;
    DEBUG_ENTER("JOIN::exec");

    ...

    result->send_fields((procedure ? curr_join->procedure_fields_list :
                                *curr_fields_list),
                        Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF);
    error= do_select(curr_join, curr_fields_list, NULL, procedure);
    thd->limit_found_rows= curr_join->send_records;
    thd->examined_row_count= curr_join->examined_rows;
}

```

还有一个值得注意的函数调用。请注意代码语句result->send_fields()。这个函数的名字恰如其分地描述了它的行为：把字段（列）的标题发送给客户端。那么，把查询结果发送给客户端的是哪一个方法呢？别着急，我将在本书的第4章向读者详细介绍那些方法。thd->limit_found_rows=和thd->examined_row_count赋值语句，负责把有关记录的计数值保存到THD类。再去看看do_select()函数。

代码清单3-13是do_select()函数的浓缩版本，你可以从中看到一些大事件。请注意最后一条以黑体字突出显示的代码语句。join->result->send_eof()函数将把eof（end-of-file，文件结束）标记发送到客户。问题又来了：查询结果哪儿去了？查询结果是在sub_select()函数里生成的，你马上就会看

到该函数。

代码清单3-13 do_select()函数

```
static int
do_select(JOIN *join, List<Item> *fields, TABLE *table, Procedure *procedure)
{
    int rc= 0;
    enum_nested_loop_state error= NESTED_LOOP_OK;
    JOIN_TAB *join_tab;
    DBUG_ENTER("do_select");

    ...

    error= sub_select(join, join_tab, 0);

    ...

    if (join->result->send_eof())

    ...
}
```

现在看一下代码清单3-14，它是sub_select()函数的一个浓缩版本。这段代码首先对JOIN类记录进行了初始化。join_init_read_record()函数对名为JOIN_TAB的结构里的记录进行读操作之前的初始化，它把一个名为READ_RECORD的类填充到read_record成员变量里。READ_RECORD类包含从表读取的元组。在这个函数的内部是存储引擎子系统的抽象层。第7章将对存储引擎进行专题讨论；向大家详细介绍如何建立存储引擎。对表进行了初始化之后，系统将开始顺序读取有关的记录，每次读取一条记录，直到把所有的记录都读取完毕。

代码清单3-14 sub_select()函数

```
enum_nested_loop_state
sub_select(JOIN *join, JOIN_TAB *join_tab, bool end_of_records)
{
    ...

    READ_RECORD *info= &join_tab->read_record;

    if (join->resume_nested_loop)
    {
        ...
    }
    else
    {
```

```

...

join->thd->row_count= 0;

error= (*join_tab->read_first_record)(join_tab);
rc= evaluate_join_record(join, join_tab, error, report_error);
}

while (rc == NESTED_LOOP_OK)
{
    error= info->read_record(info);
    rc= evaluate_join_record(join, join_tab, error, report_error);
}

...
}

```

注解 代码清单3-14里的代码比你此前见过的代码清单更为浓缩。主要原因是sub_select()函数使用了许多高级编程技术，如递归、函数指针重定向等。不过，就示例查询命令而言，这段代码所体现出来的概念是清楚的。

对表达式进行求值和对关系操作符进行处理的工作是在JOIN类里完成的。查询结果被处理完毕之后，将被传输到客户端。此后，控制权返回到do_select()函数^①，它将把eof (end-of-file，文件结束)标记发送给客户端以表明“没有查询结果了”。表3-6列出了与查询执行有关的源代码文件。

表3-6 查询执行

源代码文件	说 明
/sql/sql_select.cc	包含着一些优化函数和一些数据检索算法的具体实现

希望这次旅行能够满足大家的好奇心，更希望它能让读者体会到一个世界级数据库系统的复杂性。如果你对查询处理的基本流程有什么不清楚的地方，随时可以再做一次这样的旅行。在接下来的小节里，将讨论几个比较重要的类和结构。

3.2.8 辅助库

MySQL的源代码树里有许许多多额外的库。长期以来，MySQL AB公司一直在勤奋地对那些用来访问所支持的操作系统和硬件设备的常用例程进行封装和优化。这些辅助库里的大部分函数都是操作系统调用或硬件访问调用的打包器。这些辅助库使得程序员在编写代码时不必关心具体的软硬件平台，不必学习和编写专用的代码。这些辅助库提供的功能/数据结构包括：字符串处理、散列表、链接列表、内存分配等。表3-7列出了几个常用的辅助库。

^① 原书这里是sub_select()，有误。——译者注

提示 如果你想知道你要使用的例程是否有一个相应的库，最好的方法是用一个文本搜索工具对/mysys子目录里的源代码文件进行一次搜索。绝大多数打包器函数都有一个与相应的原始函数相类似的名字。比如说，my-alloc.c实现了malloc打包器。

表3-7 辅助库

源代码文件	功能/数据结构
/mysys/array.c	数组操作
/mysys/hash.h和/mysys/hash.c	散列表
/mysys/list.c	链接列表
/mysys/my_alloc.c	内存分配
/strings/*.c	基本的内存和字符串处理例程
/mysys/string.c	字符串操作
/mysys/my_pthread.c	线程

3

3.2.9 重要的类和结构

MySQL源代码里的许多类和结构是这个系统能取得如此成功的关键因素。要想真正掌握MySQL的源代码，就必须对所有比较重要的类和结构了如指掌。知道哪些信息保存在哪个类里或是哪个结构保存着哪些信息，可以帮助你更好地把做出的修改集成到MySQL系统里去。接下来的几个小节将向大家介绍几个非常关键的类和结构。

1. ITEM_类

ITEM_类是MySQL的每一个子系统都要用到的东西。之所以称它为ITEM_类，是因为从ITEM基类派生出了许多子类甚至是孙类。这些派生类被用来存储和处理MySQL系统里的许多种数据，其中包括：参数（如WHERE子句里的参数）、标识符、时间、字段、函数、数值、字符串等。代码清单3-15是ITEM基类的浓缩版本。ITEM基类的定义在/sql/item.h源代码文件里，它的实现代码在/sql/item.cc源代码文件里。ITEM类的子类有各自的定义文件和实现文件，那些文件的名字与它们所封装的数据保持一致。比如说，函数类的定义在/sql/item_func.h文件里，它的实现代码在/sql/item_func.cc文件里。

代码清单3-15 ITEM_类

```
class Item {
    Item(const Item &);      /* Prevent use of these */
    void operator=(Item &);
public:
    static void *operator new(size_t size)
    { return (void*) sql_alloc((uint) size); }
    static void *operator new(size_t size, MEM_ROOT *mem_root)
    { return (void*) alloc_root(mem_root, (uint) size); }
    static void operator delete(void *ptr, size_t size) { TRASH(ptr, size); }
    static void operator delete(void *ptr, MEM_ROOT *mem_root) {}
}
```



```

enum Type {FIELD_ITEM= 0, FUNC_ITEM, SUM_FUNC_ITEM, STRING_ITEM,
            INT_ITEM, REAL_ITEM, NULL_ITEM, VARBIN_ITEM,
            COPY_STR_ITEM, FIELD_AVG_ITEM, DEFAULT_VALUE_ITEM,
            PROC_ITEM, COND_ITEM, REF_ITEM, FIELD_STD_ITEM,
            FIELD_VARIANCE_ITEM, INSERT_VALUE_ITEM,
            SUBSELECT_ITEM, ROW_ITEM, CACHE_ITEM, TYPE HOLDER,
            PARAM_ITEM, TRIGGER_FIELD_ITEM, DECIMAL_ITEM,
            XPATH_NODESET, XPATH_NODESET_CMP,
            VIEW_FIXER_ITEM};

...
/*
    str_values's main purpose is to be used to cache the value in
    save_in_field
*/
String str_value;
my_string name;      /* Name from select */
/* Original item name (if it was renamed)*/
my_string orig_name;
Item *next;
uint32 max_length;
uint name_length;      /* Length of name */
uint8 marker, decimals;
my_bool maybe_null;     /* If item may be null */
my_bool null_value;     /* if item is null */
my_bool unsigned_flag;
my_bool with_sum_func;
my_bool fixed;          /* If item fixed with fix_fields */
my_bool is_autogenerated_name; /* indicate was name of this Item
                                autogenerated or set by user */

DTCollation collation;

// alloc & destruct is done as start of select using sql_alloc
Item();
/*
    Constructor used by Item_field, Item_ref & aggregate (sum) functions.
    Used for duplicating lists in processing queries with temporary
    tables
    Also it used for Item_cond_and/Item_cond_or for creating
    top AND/OR structure of WHERE clause to protect it of
    optimisation changes in prepared statements
*/
Item(THD *thd, Item *item);
virtual ~Item()
{
#ifdef EXTRA_DEBUG
    name=0;
#endif
} /*lint -e1509 */

```

```

void set_name(const char *str, uint length, CHARSET_INFO *cs);
void rename(char *new_name);
void init_make_field(Send_field *tmp_field, enum enum_field_types type);
virtual void cleanup();
virtual void make_field(Send_field *field);
Field *make_string_field(TABLE *table);

...
};

```

2. LEX结构

LEX结构是SQL命令在MySQL系统内部的表示形式。SQL命令的各组成部分——字段、表、表达式等全都井井有条地存储在LEX结构里。

在对一条查询命令进行解析的时候，解析器会把识别出来的每一个基本元素填充到相关的LEX结构里。在离开解析器的时候，LEX结构将包含着对查询进行优化和执行所必须的每一样东西。代码清单3-16是LEX结构的浓缩版本，该结构是在/sql/lex.h源代码文件里定义的。

代码清单3-16 LEX结构

```

typedef struct st_lex
{
    uint yylineno, yytoklen;      /* Simulate lex */
    LEX_YSTYPE yylval;
    SELECT_LEX_UNIT unit;        /* most upper unit */
    SELECT_LEX select_lex;       /* first SELECT_LEX */
    /* current SELECT_LEX in parsing */
    SELECT_LEX *current_select;
    /* list of all SELECT_LEX */
    SELECT_LEX *all_selects_list;
    const uchar *buf;            /* The beginning of string, used by SPs */
    const uchar *ptr, *tok_start, *tok_end, *end_of_query;

    /* The values of tok_start/tok_end as they were one call of yylex before */
    const uchar *tok_start_prev, *tok_end_prev;

    char *length, *dec, *change, *name;
    char *help_arg;
    char *backup_dir;            /* For RESTORE/BACKUP */
    char* to_log;                 /* For PURGE MASTER LOGS TO */
    char* x509_subject, *x509_issuer, *ssl_cipher;
    char* found_semicolon;       /* For multi queries - next query */
    String *wild;
    sql_exchange *exchange;
    select_result *result;
    Item *default_value, *on_update_value;
    LEX_STRING comment, ident;
    LEX_USER *grant_user;
    XID *xid;
    gptr yacc_yyss, yacc_yyvs;

```

```

THD *thd;
CHARSET_INFO *charset;
TABLE_LIST *query_tables; /* global list of all tables in this query */

...
} LEX;

```

3. NET结构

NET结构保存着MySQL服务器与客户端进行通信所需要的所有信息。代码清单3-17是NET结构的浓缩版本。buff成员变量用来存放原始的通信数据包（SQL命令就包含在这些数据包里）。在后面的有关章节里，你将会看到许多用来填写、读取、发送/接收数据的辅助函数，下面就是其中的两个：

- my_net_write()函数，把数据包从NET结构写到网络协议。
- my_net_read()函数，把数据包从网络协议读入NET结构。

你可以在/include/mysql_com.h文件里找到全套的网络通信函数。

代码清单3-17 NET结构

```

typedef struct st_net {
#if !defined(CHECK_EMBEDDED_DIFFERENCES) || !defined(EMBEDDED_LIBRARY)
    Vio* vio;
    unsigned char *buff,*buff_end,*write_pos,*read_pos;
    my_socket fd; /* For Perl DBI/dbd */
    unsigned long max_packet,max_packet_size;
    unsigned int pkt_nr,compress_pkt_nr;
    unsigned int write_timeout, read_timeout, retry_count;
    int fcntl;
    my_bool compress;
    /*
     * The following variable is set if we are doing several queries in one
     * command ( as in LOAD TABLE ... FROM MASTER ),
     * and do not want to confuse the client with OK at the wrong time
     */
    unsigned long remain_in_buf,length, buf_length, where_b;
    unsigned int *return_status;
    unsigned char reading_or_writing;
    char save_char;
    my_bool no_send_ok; /* For SPs and other things that do multiple stmts */
    my_bool no_send_eof; /* For SPs' first version read-only cursors */
    /*
     * Set if OK packet is already sent, and we do not need to send error
     * messages
     */
    my_bool no_send_error;
    /*
     * Pointer to query object in query cache, do not equal NULL (0) for
     * queries in cache that have not stored its results yet
     */
#endif
}

```

```

char last_error[MYSQL_ERRMSG_SIZE], sqlstate[SQLSTATE_LENGTH+1];
unsigned int last_errno;
unsigned char error;
gptr query_cache_query;
my_bool report_error; /* We should report error (we have unreported error) */
my_bool return_errno;
} NET;

```

4. THD类

在前面介绍MySQL源代码的时候，我曾经多次提到过THD类。事实上，每条连接都恰好对应着一个THD对象。对MySQL这样的多线程系统来说，线程类是线程执行成功的最大关键；从实现访问控制到把查询结果返回给客户端，每一个操作都离不开它。在MySQL服务器里，几乎每一个子系统或函数都要用到THD类。代码清单3-18是THD类的浓缩版本。多花点儿时间去浏览一下它里面的成员变量和方法。正如你看到的那样，这是一个非常大的类（代码清单3-18省略了许多方法）。这个类的定义在/sql/sql_class.h源代码文件里，它的实现代码在/sql/sql_class.cc源代码文件里。

代码清单3-18 THD类

```

class THD : public Statement,
            public Open_tables_state
{
public:

    ...

    String packet;          // dynamic buffer for network I/O
    String convert_buffer;   // buffer for charset conversions
    struct sockaddr_in remote; // client socket address
    struct rand_struct rand;  // used for authentication
    struct system_variables variables; // Changeable local variables
    struct system_status_var status_var; // Per thread statistic vars
    THR_LOCK_INFO lock_info;    // Locking info of this thread
    THR_LOCK_OWNER main_lock_id; // To use for conventional queries
    THR_LOCK_OWNER *lock_id;    // If not main_lock_id, points to
                                // the lock_id of a cursor.
    pthread_mutex_t LOCK_delete; // Locked before thd is deleted

    ...

    char *db, *catalog;
    Security_context main_security_ctx;
    Security_context *security_ctx;

    ...

    enum enum_server_command command;
    uint32 server_id;
    uint32 file_id; // for LOAD DATA INFILE

    ...
}

```



```

const char *where;
time_t      start_time,time_after_lock,user_time;
time_t      * connect_time,thr_create_time; // track down slow pthread_create
thr_lock_type update_lock_default;
delayed_insert *di;

...

table_map  used_tables;

...

ulong      thread_id, col_access;

...

inline time_t query_start() { query_start_used=1; return start_time; }
inline void  set_time()    { if (user_time) start_time=time_after_lock=user_time;
                           else time_after_lock=time(&start_time); }
inline void  end_time()    { time(&start_time); }
inline void  set_time(time_t t) { time_after_lock=start_time=user_time=t; }

...
};

```

现在，在陪伴大家经历了一次MySQL源代码之旅，并介绍了一些重要的类和结构之后，将把讨论重点转到如何帮助大家根据自己的具体需要去修改MySQL系统上来。在再次回到MySQL源代码之前，我想先和大家探讨一下软件开发工作中编程风格和文档维护方面的问题。

3.3 编程指导

如果你对前面那些源代码的格式不习惯，那很可能是因为你的编程风格与那些源代码作者的风格不一样。像MySQL这样的大型软件是许多程序员共同努力的成果，他们每个人都有自己的风格。如果放任自流，代码很快就会变成一堆杂乱的语句。为了避免这种问题，MySQL AB公司以各种形式推出了编程指南。不过，在研究MySQL源代码的时候，我发现（你们迟早也会遇到）有个别程序员没有遵守MySQL AB公司的编程指南。往好处想，这大概是因为那份指南随着时间的推移而发生了变化的缘故，这对一个大型软件项目来说并不稀奇。为什么有人不遵守编程指南，这与本书的论题无关，我只想告诉大家绝大多数程序员都是按照那份指南进行编程的。更重要的是，MySQL AB公司也希望你们能遵守。

那份编程指南收录在 *MySQL Internals Manual*（MySQL 内部原理手册）里，可以从 <http://dev.mysql.com/doc> 上找到该手册。该编程指南列在这本手册的第2章，它对使用C/C++语言为MySQL服务器编写代码时“允许做什么”和“不允许做什么”做了详细的规定。我从中摘选了一些最重要的规定并在接下来的几个段落里进行了汇总。

3.3.1 总体指导

那份指南特别强调你应该按照尽可能优化的方式来编写代码。这个原则与敏捷开发（agile development, AD）方法学是相抵触的，AD方法学的基本思路是“需要什么，编写什么；细化和优化在复查阶段再说”。如果你已经习惯于AD方法，在向MySQL源代码库提交你的修改之前千万要进行复查。

另一个非常重要的总体原则是：不要直接使用最底层的API或操作系统调用，应该到相关的库里寻找相应的打包器函数来编程。那些打包器函数都经过了MySQL AB公司的优化，在速度和安全性方面都有保证。比如说，不应该使用C语言中的malloc()函数，应该使用sql_alloc()或my_alloc()函数。

所有的代码行必须少于80个字符。如果你的语句在一个代码行里容纳不下，可以延续到下一行，但必须把参数垂直对齐或是对续行进行统一的缩进。

注释必须使用标准的C语言注释，例如/* this is a comment */。在此前提下，程序员可以在代码的任何位置插入注释。

提示 MySQL编程指南特别强调：不要使用C++的// comment选项风格的注释。

3.3.2 文档

源代码中的首选自然语言是英语。这包括所有的变量名、函数名、常数和注释。编写和维护MySQL源代码的程序员主要分布在欧洲和美国。选择英语作为源代码中的首选自然语言主要是因为美国的计算机科技水平比较高，影响也比较大；另一个因素是英语在许多欧洲国家都是小学或中学开设外语课程的首选。

在编写函数的时候，你应该在开头部分用一个注释块对基本用途、参数、预期的返回值等做出说明。注释块的内容应该分成段落，段落的标题用全大写字母给出。你应该在注释后面的第一行给函数起一个简短的描述性的名字，并且至少要包括节、提要、描述和返回值。还可以包括可选的节，如WARNING（警告）、NOTES（注解）、SEE ALSO（另见）、TODO（完成事项）、ERRORS（出错处理）和REFERENCED_BY（引用情况）。这几个节及其内容描述如下。

- ❑ SYNOPSIS（必需的）——对函数的流程和控制情况进行概述。必须把这个函数的基本算法讲清楚。这可以帮助别人一眼就能看出这个函数是干什么用的。这个节还包括对所有参数的描述（输入参数用“IN”表示，输出参数用“OUT”表示，取值会发生变化的参数用“IN/OUT”表示）。
- ❑ DESCRIPTION（必需的）——对函数基本情况的描述。包括对函数的用途和用法的简要说明。
- ❑ RETURN VALUE（必需的）——对所有可能出现的返回值以及其含义进行说明。
- ❑ WARNING——如果这个函数有一些不同寻常的副作用，必须在这个节里加以说明。
- ❑ NOTES——可以把你认为比较重要的其他信息写在这个节里。
- ❑ SEE ALSO——如果这个函数需要与其他函数配合工作，需要以另一个函数的输出作为输入，或者是需要按照某种特定的顺序被另一个函数调用，就应该在这个节里写清楚。
- ❑ TODO——如果这函数有尚未完成的功能，应该在这个节里加以说明。在你完成它们之后，千万记得从这个节里把有关的说明删掉。我有时会忘记这样做——结果是忙了半天才发现自己

其实已经完成了TODO工作。

- ❑ **ERRORS**——如果你的函数有特殊的出错处理功能，请在这个节里加以说明。
- ❑ **REFERENCED BY**——在这个节里对你的函数与其他函数或对象的关系进行说明。比如说，你的函数会在何时调用另一个函数，这个函数是不是另一个函数的打包器或内含函数，这个函数是不是一个友方法，或者不是一个虚拟函数等。

提示 对于只有几行代码的小函数，MySQL AB公司建议不必提供如上所述的注释块，但我建议你为所有的函数都编写注释块。等你研究MySQL源代码，并遇到大量没有或只有几句文档的小（以及一些比较大的）函数时，就会明白这个建议的好处。

代码清单3-19给出了一个比较完整的函数注释块示例。

代码清单3-19 函数注释块示例

```
/*
Find tuples by key.

SYNOPSIS
find_by_key()
string key          IN      A string containing the key to find.
Handler_class *handle IN    The class containing the table to be searched.
Tuple *             OUT    The tuple class containing the key passed.

Uses B Tree index contained in the Handler_class. Calls Index::find()
method then returns a pointer to the tuple found.

DESCRIPTION
This function implements a search of the Handler_class index class to find
a key passed.

RETURN VALUE
SUCCESS (TRUE)      Tuple found.
!= SUCCESS (FALSE)  Tuple not found.

WARNING
Function can return an empty tuple when a key hit occurs on the index but
the tuple has been marked for deletion.

NOTES
This method has been tested for empty keys and keys that are greater or
less than the keys in the index.

SEE ALSO
Query::execute(), Tuple.h

TODO
```

```
* Change code to include error handler to detect when key passed in exceeds
the maximum length of the key in the index.
```

```
ERRORS
```

```
-1          Table not found.
1          Table locked.
```

```
REFERENCED_BY
```

```
This function is called by the Query::execute() method.
```

```
*/
```

3.3.3 函数和参数

我之所以把它们单独提出来加以讨论，是因为MySQL源代码里有一些相互矛盾的做法。如果你以MySQL源代码作为代码排版指南的话，你很可能在不知不觉中偏离编程指南。函数和它们的参数必须按照参数垂直对齐的方式对齐。在定义函数和调用函数的时候都应该如此。类似的，在声明变量的时候也应该把它们垂直对齐。对齐时的缩进量并不重要，重要的是必须让有关参数或变量保持垂直对齐的样子。你还应该给每一个变量加上一个“行注释”（line comment）。行注释应该从第49列开始写起，但不得超过每行80个字符。如果某个变量的注释超过了80列，应该把那条注释单独写在一行上。代码清单3-20给出了一个对函数、变量和参数进行对齐的例子。

代码清单3-20 对变量、函数和参数进行对齐的例子

```
int    var1;                /* comment goes here */
long   var2;                /* comment goes here too */
/* variable controls something of extreme interest and is documented well */
bool   var3;

return_value *classname::classmethod(int var1,
                                       int var2
                                       bool var3);

if (classname->classmethod(myreallylongvariablename1,
                          myreallylongvariablename2,
                          myreallylongvariablename3) == -1)
{
    /* do something */
}
```

警告 如果你是在Windows平台上进行编程的，请注意你使用的文本编辑器的换行功能是否符合MySQL编程指南的规定。Windows平台上的绝大多数编辑器都使用一个CRLF序列（/r/n）作为换行符，但MySQL AB公司要求你使用单个的LF（/n）而不是CRLF。这是在Windows平台上创建的文件与在Unix/Linux平台上创建的文件之间最常见的不兼容之处。因此，如果你使用的是Windows，请检查你的编辑器并对其配置做必要的调整。

3.3.4 命名约定

MySQL AB公司推荐的变量命名约定是：挑选有意义的名字，全部使用小写字母，以下划线分隔各个单词而不是使用首字母大写的单词序列。但这里有一个例外：类的名字必须使用首字母大写的单词序列。此外，枚举数据类型应该带有enum_前缀；所有的结构和定义必须以全大写字母命名。代码清单3-21给出了一个符合上述命名约定的例子。

代码清单3-21 命名约定示例

```
class My_classname;
int my_integer_counter;
bool is_saved;

#define CONSTANT_NAME 12;

int my_function_name_goes_here(int variable1);
```

3.3.5 分隔与缩进

MySQL编程指南建议：每级缩进都应该偏移两个字符位置，不允许使用制表符。如果你的编辑器允许，你应该改变它的默认行为或关闭它的自动排版功能，并把所有的制表符替换为两个空格。在使用诸如Doxygen（我稍后再做讨论）之类的文档工具或使用某个代码行解析工具在文本里查找字符串的时候尤其要注意这一点。

在对标识符和操作符进行排版的时候，变量与操作符之间不应该有任何空格，操作符与操作数之间应该有一个空格（加在操作符的右边）。类似的，在函数里，左括号的后面不应该有任何空格，参数之间应该有一个空格，最后一个参数与右括号之间不应该有任何空格。最后，你应该在以下位置加上一个空行来分隔前、后两个部分：变量声明部分与控制代码部分之间、控制代码部分与函数调用语句之间、注释块与其他代码之间、函数与其他声明之间。代码清单3-22给出了一段符合上述规则的代码，其中包括一条赋值语句、一个函数调用和一条控制语句。

代码清单3-22 分隔与缩进

```
return_value= do_something_cool(i, max_limit, is_found);
if (return_value)
{
    int var1;
    int var2;

    var1= do_something_else(i);

    if (var1)
    {
        do_it_again();
    }
}
```

在MySQL的源代码里，花括号的对齐方式也不总是一样的。MySQL编程指南规定：花括号应该

与它上面的控制代码对齐。就像我在前面所有的示例代码里展示的那样。不过，如果你需要缩进到下一级，应该按照代码缩进规则（偏移两个空格）来缩进花括号。此外，如果某个代码块里只有一行代码，也可以不使用花括号。

MySQL编程指南对switch语句里花括号的缩进情况是这样规定的：switch语句的最外层左花括号应该写在switch条件的后面，最外层右花括号应该与switch关键字垂直对齐；各case语句应该与switch关键字垂直对齐。代码清单3-23给出了一个这样的例子。

代码清单3-23 switch语句示例

```
switch (some_var) {
case 1:
    do_something_here();
    do_something_else();
    break;
case 2:
    do_it_again();
    break;
}
```

注解 代码清单3-23里的最后一条break语句并非必不可少。我喜欢在代码里加上它，因为我觉得这样才算完整。

3.3.6 文档工具

用一个自动化的文档生成器来分析源代码是一个很有用的办法。这类工具可以快速扫描大量的源代码并生成各种函数清单、类清单、结构清单等，你可以利用这些清单快速找到想要的东西。MySQL的源代码在操作和处理数据时需要依赖许多重要的数据结构，如果手里有一份这些数据结构的清单，想找什么东西就容易多了。

Doxygen就是一个这样的文档工具。Doxygen也是一个采用GPL许可证发行的开源软件。在启动之后，Doxygen将读取有关的源代码并生成一组非常易读的HTML文件；它可以从源代码里把函数前面的注释提取出来并列函数调用指令。Doxygen可以阅读和处理用C、C++、Java和另外几种编程语言写的源代码。在分析一个像MySQL这样的复杂系统时，Doxygen是一个非常有用的工具——有些基本的库函数会在源代码里的好几百处地方被调用，用你的肉眼去寻找根本不现实。

Doxygen适用于Unix和Windows平台。这个工具的Linux源代码版本可以从Doxygen网站（www.stack.nl/~dimitri/doxygen）下载。

把有关文件下载回来之后，按照安装指示（也在该网站上）安装。Doxygen使用配置文件来生成输出的外观和体验以及输入的内容。下面这条命令将生成一个默认配置文件：

```
doxygen -g -s /path_to_new_file/doxygen_config_filename
```

Doxygen生成的文档将被保存到上面这条命令所指定的文件路径。有了一个默认配置文件，就可以编辑这个文件并根据具体需要改变各有关参数了。对有关选项及其参数的详细说明见Doxygen的随机文档。需要改变的设置主要有：准备处理的文件夹、项目名和其他与项目有关的设置。把配置文件

修改满意之后，发出下面这条命令就可以生成文档了：

```
doxygen </path_to_new_file/Doxygen_config_filename>
```

注意 根据你的设置，Doxygen可能会运行很长时间。如果你想让Doxygen尽快生成文档，请避免使用高级图形化命令。

Doxygen的最新版本可以在Windows系统上运行并提供一个GUI。这个GUI包括一个引导你创建一个基本配置文件的向导和一个供你设置各种参数的专家模式，还允许你从多个配置文件中挑选一个来加载。我发现，通过向导界面生成的输出可以满足大多数情况的需要。

建议大家在一头扎进源代码之前先花些时间运行Doxygen并仔细研究一些它的输出文件；这可以让你节约大量的查找时间。Doxygen生成的文档值得贴在你显示器旁边的墙上或是剪贴到你的工作记录本里。图3-3是一个Doxygen生成文档的示例。

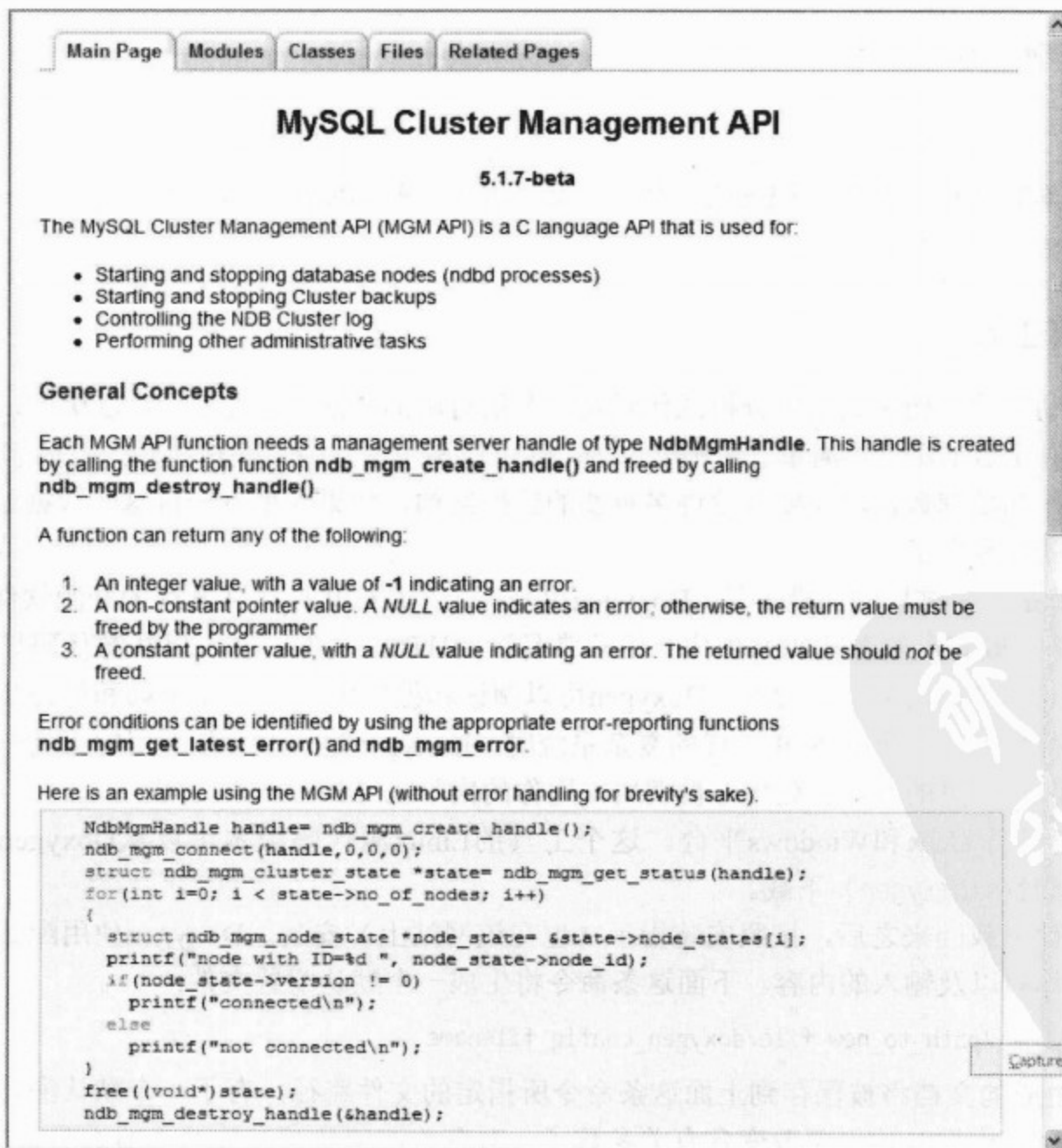


图3-3 Doxygen生成的文档

3.3.7 保持工作记录的习惯

许多程序员会随时记录自己项目的进展情况。虽然在详细程度上会有所差异，但绝大多数程序员都会在开会和打电话时把要点记下来形成一份书面的记录。如果你还没有养成这种习惯，建议从现在开始试一试。工作记录本是我在工作中的重要工具之一。应该承认，把东西写下来是有点儿麻烦，如果记的东西太多了，查起来也不是很方便。就拿我的记录本来说，手绘的草图、电子邮件地址、剪贴下来的重要资料，这些东西让我的记录本更像一个剪贴簿。不过，这个小本子的作用可不容小觑。

在研究MySQL源代码的时候，准备一个笔记本会对你有很大的帮助。你可以把每一个发现记在这个笔记本里，还可以把你领悟到的东西、重要的设计决策、重要文档的关键内容、突然闪现的灵感等随时记录下来。随着时间的推移，你积累下来的东西将成为一笔可观的财富（我的一位朋友把它称为“纸脑”！），它不仅可以帮助你回想起当初，更可以启发你现在的思路。如果你已经有了这个好习惯，将发现这种笔记本用不着过分追求条理整齐。有些程序员（包括我在内）喜欢使用硬皮本，但它没法重新编排（除非动用剪刀和胶水）；另一些程序员则喜欢使用活页本并定期对之进行整理。如果你决定使用一个硬皮本，请考虑为你记下来的东西建立一个“实时的”索引。

提示 如果你的笔记本没有页码，请花几分钟的时间给每一页加上一个页码。

建立实时索引的办法有很多。可以把你认为有意思的关键字写在页面的顶部或页边的特定位置，这可以让你快速浏览你的笔记本并找到想查找的东西。在这里，“实时”的意思是说索引能随时添加引用。我发现的建立实时索引的最好办法是，把笔记本里的术语和相关的页码用一个电子表格软件列成表格，每隔一个星期左右把它打印出来贴到笔记本上。只要你觉得方便，怎么贴都行。这样一来，我可以随时根据手头的工作增减那个表格里关键字和/或调整它们的先后顺序，让它更便于使用。

我再次建议读者考虑使用一个工作记录本。在你向顶头上司汇报工作进展情况的时候，这个小本子可以让你胸有成竹。如果你被问到的事情是在六个月甚至更早之前发生的，这小本子的作用就更大了。

3.3.8 追踪变化

只要你写出来的代码不太容易看懂，就应该加上注释。比如说，代码`if (found)`的意思是个程序员就应该能看懂，没必要加注释；但`if (func_call_17(i, x, lp))`就不一样了，应该在注释里加以说明。程序员都希望自己写出来的代码易读易懂，但这在某些时候是不可能的。在调用各种底层库函数时尤其如此。有些函数从名字上根本看不出是干什么用的，即使能从参数表上看出点儿眉目，也只能作为猜测而非结论。总之，在编写代码的时候，如果能把这类情况随时记录在案，你将为你本人和阅读你代码的其他程序员减少很多烦恼。

在添加注释的时候，可以选择行内注释、单行注释或多行注释。行内注释应该从第49列开始写起，但不允许超过第80列。单行注释应该与它所说明的那条语句对齐，也不允许超过第80列。类似的，多行注释应该与它们所说明的语句对齐；不应该超过第80列；注释块的首尾标记应该单独占一行。代码清单3-24体现了这些概念。

代码清单3-24 注释行/注释块的排版示例

```

if (return_value)
{
    int    var1;           /* comment goes here */
    long   var2;           /* comment goes here too */

    /* this call does something else based on i */
    var1= do_something_else(i);

    if (var1)
    {
        /*
         * This comment explains
         * some really interesting thing
         * about the following statement(s).
         */
        do_it_again();
    }
}

```

提示 不要使用连续的*来突出某个代码段。这容易分散读者的注意力并让代码显得很杂乱。更麻烦的是，把这些东西对齐需要花很多时间，尤其是在你再次编辑你的注释内容时。

如果你是使用源代码控制软件BitKeeper来修改MySQL源代码的，用不着担心如何追踪你的修改的问题。BitKeeper提供了好几种办法可以让你查明哪些修改是你做的、哪些修改是别人做的。可是，如果你没有使用BitKeeper，就有可能说不清哪些修改是你做的。万一你是在没做备份的情况下直接修改了某个函数（这种习惯很不好，但总有人这样做），把你做的修改和原来的代码区分开将变得非常困难。如果你有一个工作记录本，这时候可就帮上大忙了。但这里还有一个更好的办法。

你可以在做的修改之前和之后加上一些注释，用这些注释把你的代码标识出来。比如说，可以在代码段的开头加上一条/* BEGIN CAB MODIFICATION */注释，在代码段的末尾加上一条/* END CAB MODIFICATION */注释。这两条注释就像一对括号把你的代码段括在中间，如果日后需要找出它们，用你最喜欢的文本编辑工具做一次搜索就行了。代码清单3-25给出了一个这样的例子。

代码清单3-25 用注释把你的代码与MySQL源代码区分开

```

/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds my revision note to the MySQL version number. */
/* original code: */
/*strmov(end, "."); */
strmov(end, "-CAB Modifications");
/* END CAB MODIFICATION */

```

虽然这个例子是虚构的，但有两点希望大家注意：一是写明了进行修改的理由；二是把原来的语句保留在了注释里。这个技巧不仅可以让你快速找到你的修改，还为日后的故障诊断工作提供了方便。

如果你做的修改仅限于你本人或你的公司使用，不想与MySQL AB公司共享，这个技巧也非常有

用。如果你不共享修改，当MySQL AB公司推出一个新版本时，如果你打算升级到新版本，就必须再次修改有关的源代码。此时，源代码里的注释块可以让你迅速找到需要修改的文件，还可以让你知道都要修改些什么。可话又说回来了，MySQL每发布一个新版本就做一遍修改确实很麻烦。如果你真的创造了一些新功能的话，很可能会为了避免这种麻烦而与MySQL AB公司共享它。

注意 在通过BitKeeper使用MySQL源代码的时候，这个技巧虽然没有被禁止，但也不受鼓励。原因很简单：其他程序员也许会把你的注释全部删除。因此，从对自己和对别人都负责的角度出发，你应该只在你将不与任何人分享修改的情况下才使用这个技巧。

3.4 第一次构建系统

你已经看到了MySQL源代码的内部工作情况，也沿着一条典型的查询浏览了一遍那些源代码，现在是来转动一下车轮的时候了。如果你早就在使用MySQL，现在只是通过这本书去学习更多关于MySQL源代码的知识和如何修改的话，可以跳过这一节。

在动手之前，我还有一个建议：不仅要下载MySQL源代码，还要为你的平台下载并安装相应的可执行版本。备有一个二进制可执行版本非常重要。如果你实验期间发生了问题，可以把系统迅速恢复成原来的样子。在没有参照点的情况下，对经过修改的MySQL源代码进行调试和纠错是一个非常大的挑战。在你遇到一个非常困难的调试问题时，“原装的”可执行文件可以为你节约大量的时间。我将在第5章对调试问题做进一步讨论。只要你遇到了难以解决的系统问题，随时可以通过重新安装二进制版本的办法把MySQL系统恢复到正常状态。

编译源代码很容易。如果你正在使用Linux，打开一个命令shell，进入MySQL源代码树的根目录，然后执行configure、make和make install命令。

注解 如果正在使用Linux而用来编译MySQL源代码的配置文件不存在，那么需要使用BUILD子目录里的某个平台脚本来生成该文件。比如说，如果你想用debug为Pentium级机器创建配置文件，就需要从源代码树的根目录使用命令./BUILD/compile-pentium-debug。在创建出这个文件后，你就可以使用./configure、make和make install命令来编译MySQL服务器了。

配置脚本将检查系统的依赖关系，并创建一个相应的制作文件。make和make install命令将完成建立系统和进行安装的工作。绝大多数程序员在建立MySQL源代码时都执行命令。如果这是你第一次编译MySQL源代码，可能需要改变某些文件的所有者（如果你不是root用户的话）并对用户组情况做出必要的调整——详见《MySQL参考手册》的2.81节（<http://dev.mysql.com/doc/refman/5.1/en/quick-install.html>）。下面是在Linux平台上第一次编译MySQL源代码的一个典型编译过程：

```
%> groupadd mysql
%> useradd -g mysql mysql
%> gunzip < mysql-VERSION.tar.gz | tar -xvf -
%> cd mysql-VERSION
%> ./configure --prefix=/usr/local/mysql
%> make
```

```
%> make install
%> cp support-files/my-medium.cnf /etc/my.cnf
%> cd /usr/local/mysql
%> bin/mysql_install_db --user=mysql
%> chown -R root .
%> chown -R mysql var
%> chgrp -R mysql .
%> bin/mysqld_safe --user=mysql &
```

Windows用户可以用Microsoft Visual Studio 2005来编译MySQL源代码（有些人用配有Microsoft平台开发包的Visual Studio 6.0和2005 Express Edition也取得了成功，但我发现Visual Studio 2005更加稳定）。如果你是第一次在Windows平台上编译MySQL源代码，需要在MySQL源代码树的根目录打开mysql.dsw项目工作区，把活跃项目设置为mysqld类，把项目配置设置为mysqld - Win32 nt。当你单击Build mysqld按钮时，该项目将对所有必要的库进行编译并把它们链接为指定的项目。因为这是第一次编译，需要编译很多的库，所以会花费比较长的时间，你大可以去喝杯汽水放松一下。不管你使用的是哪一种平台，作为编译结果的可执行文件都将根据你选用的编译选项放在client-release或client-debug文件夹里。如果你想运行新的可执行文件，请先结束当前服务器服务，再把有关文件复制到MySQL安装目录下的bin文件夹里，最后重新启动服务器服务即可。

注意 绝大多数编译问题都与配置不当的开发工具或缺少某个库有关。比较常见的编译问题可以在MySQL论坛上找到解决办法。

你已经编译出了一个新的二进制可执行文件（这里暂时不考虑编译失败的情况），但你将注意到的第一件事是，无法确认这个二进制可执行文件就是你编译出来的那一个！虽然可以通过查看文件创建日期的办法来确定这个可执行文件就是你创建的，但没有一种办法能从客户端确认这一点。其实办法倒不是没有，只是MySQL AB公司不推荐这种做法而已：在编译时改变MySQL的版本号，可以迅速判断出它是不是你编译的。

我们不妨假设你想一眼就认出你的修改。比如说，如果想通过客户端窗口看到信息，确认服务器就是你修改出来的那个版本，则可以通过改变版本号的办法来做到这一点。图3-4给出了一个这样的例子。

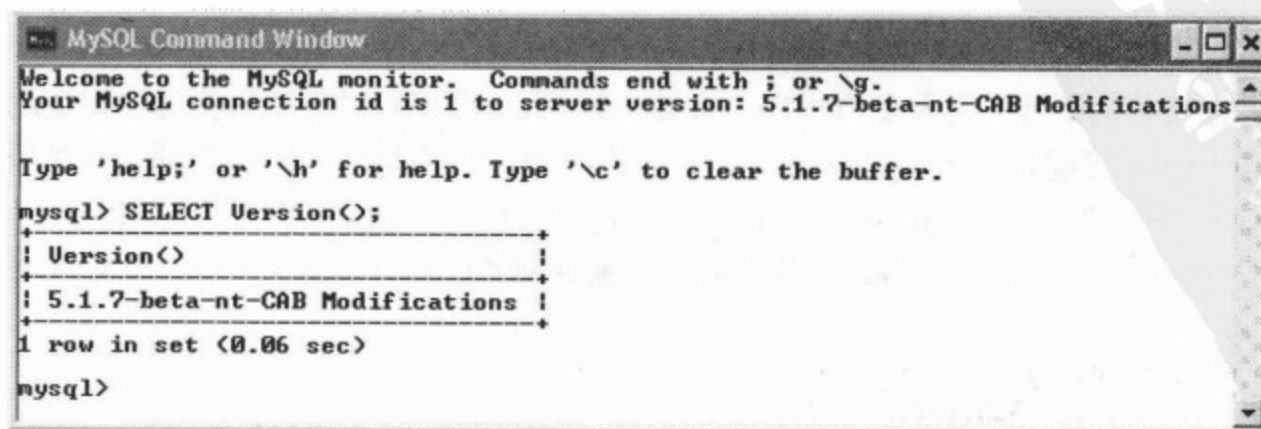


图3-4 改变版本号的MySQL命令客户端示例

注意在首部和执行SELECT VERSION();命令的结果中，返回的版本号是你编译的服务器的版本号，

后面接一个标签，这个标签是我加在字符串里的那个。如果你也想试试这个办法，只要参照代码清单3-26去修改mysqld.cpp文件里的set_server_version()函数就行了。在这个例子里，我已经把用来产生上述效果的语句用黑体字突出显示了出来，你可以根据自己的情况进行类似修改。

代码清单3-26 修改后的set_server_version()函数

```
static void set_server_version(void)
{
    char *end= strxmov(server_version, MYSQL_SERVER_VERSION,
                      MYSQL_SERVER_SUFFIX_STR, NullS);
#ifdef EMBEDDED_LIBRARY
    end= strmov(end, "-embedded");
#endif
#ifdef DEBUG_OFF
    if (!strstr(MYSQL_SERVER_SUFFIX_STR, "-debug"))
        end= strmov(end, "-debug");
#endif
    if (opt_log || opt_update_log || opt_slow_log || opt_bin_log)
        strmov(end, "-log"); // This may slow down system
    /* BEGIN CAB MODIFICATION */
    /* Reason for Modification: */
    /* This section adds my revision note to the MySQL version number. */
    strmov(end, "-CAB Modifications");
    /* END CAB MODIFICATION */
}
```

请注意我在代码清单3-26里新添加的注释，我在前面已经对如何添加这样的注释进行过介绍。这些注释可以帮你迅速找出曾经修改过的代码行。这个改动还有另一个好处：新的版本号在其他的MySQL工具（比如MySQL Administrator）里也会显示成如图3-4所示的样子。图3-5是用MySQL Administrator工具去连接一个在经过上述修改后编译出来的MySQL服务器时看到的结果。

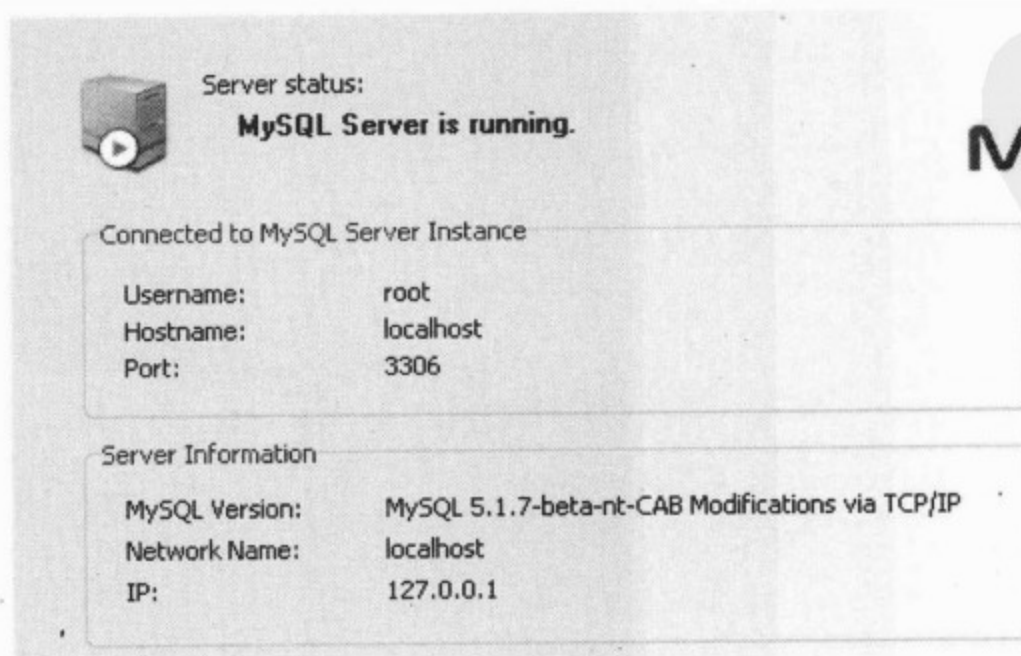


图3-5 用MySQL Administrator工具去访问修改后的MySQL服务器

注意 我刚才说过,这种做法是MySQL AB公司不认同的!如果你只是在使用MySQL进行实验或者只是在修改源代码供自己使用,按照我刚才演示的办法去做是可以的。可是,如果你是通过源代码控制工具(BitKeeper)来使用MySQL源代码的,或者如果希望你做出的修改最终能够进入MySQL源代码库,那就千万不要使用这个技巧。

3.5 小结

本章介绍了几种获得MySQL源代码的办法,可以下载源代码树的快照,可以下载GA版源代码,还可以使用BitKeeper客户软件去下载最新最好的版本。不管你选择什么办法,都可以获得并开始使用MySQL源代码。这正是开源的魅力所在!

本章最值得回味的应该是在指导下体验MySQL源代码之旅。我们跟着一个简单的查询命令,一起经历了在MySQL系统里的整个旅程,对MySQL各主要子系统和有关的源代码进行了剖析,希望这些剖析能够为你钻研MySQL源代码打下一个基础。同时希望你不会因为在编译MySQL源代码时遇到了一些问题而沮丧地将这本书扔到一边。一名优秀的开源开发人员必须具备这样一种能力:能够根据当前项目的需求对自己的软硬件环境做全面细致的诊断和调整。在遇到难题的时候千万不要气馁,更不应该放弃。解决问题是学习过程中很自然的一部分。

本章还向大家讲解了《MySQL编程指导》里的一些重要元素,并给出了一些关于代码格式和文档编制方面的例子。虽然不够完整,但本章给出的编程指导应该能让你了解MySQL AB公司希望你如何编写源代码来进行修改。如果从一开始就遵守这些简单的指导,你日后就不会有人要求你按照这些指导返工的事情发生了。

接下来的两章将探讨两个在软件开发活动中经常被人们忽视的概念。第4章将展示如何把测试驱动的开发方法学应用到探索和扩展MySQL系统的工作中。第5章将讨论如何调试MySQL源代码。



系统集成商必须克服它们正在集成的系统的限制。有时系统会缺少集成所需要的特定函数或命令。MySQL AB公司认识到了这种需要并在MySQL服务器里包括了许多可以用来添加新函数和新命令的选项。本章将向大家介绍一个为MySQL系统生成高质量扩展的关键因素。本书还将讨论软件测试并解释一些测试大型软件系统的常用实践方法，同时使用具体的例子来展示几种已被广泛接受的测试MySQL系统的方法。

4.1 背景知识

你也许会对本书这么早就用一章的篇幅来讨论软件测试问题感到奇怪。之所以要这样做，是因为我想介绍一下用来测试MySQL系统的工具和技巧，这样你在计划自己的修改时，能够从计划如何测试它们开始。这正是测试驱动开发（test-driven development）技术的精髓：根据项目要求开发并实现必要的测试，编写代码，然后立刻执行这些测试。不熟悉这个概念的人可能会感到困惑：怎样才能为还没有编写出来的代码编写测试呢？接下来的几个小节将提供一些与这个越来越流行的概念有关的背景知识，希望它们能澄清大家心中的困惑。

4.1.1 为什么要测试

只要一提到关于软件质量的问题，就会有人问到这个问题。有些学生想知道进行多少测试才行。对于那些认为测试只是在浪费时间或吹毛求疵的学生，我会给他们一个机会让他们以测试工作量最小（有时甚至是没有）的策略来完成他们的软件工程的课程项目^①。

我的学生经常向我说起他们的模块和类编写得多么巧妙，他们是如何认真地遵守了建模原则。有许多学生还使用了UML（Unified Modeling Language，统一建模语言）图来帮助自己进行开发。这些做法都很好，但测试绝不仅仅是保证你的源代码与你的模型相匹配那么简单。有不少对自己的编程技巧很自信的学生交上来的作业经常存在着一些功能方面的小毛病。虽然那些小毛病一般不至于引发致命错误或崩溃（这在开发工作中经常会遇到），但在集成和软件的运行方面经常出问题。也就是说，这些学生没能做到让他们的软件按照顾客希望的那样工作。

如果你对这种场景很熟悉，就说明你已经了解了软件测试的价值。软件测试的形式有很多，目的都是保证和控制软件的质量。在什么时候选择什么技术进行测试是软件测试学需要解决的根本问题。

^① 这通常是一个持续整个学期的大型分组项目，从项目需求分析开始。

系统集成商必须克服它们正在集成的系统的限制。有时系统会缺少集成所需要的特定函数或命令。MySQL AB公司认识到了这种需要并在MySQL服务器里包括了许多可以用来添加新函数和新命令的选项。本章将向大家介绍一个为MySQL系统生成高质量扩展的关键因素。本书还将讨论软件测试并解释一些测试大型软件系统的常用实践方法，同时使用具体的例子来展示几种已被广泛接受的测试MySQL系统的方法。

4.1 背景知识

你也许会对本书这么早就用一章的篇幅来讨论软件测试问题感到奇怪。之所以要这样做，是因为我想介绍一下用来测试MySQL系统的工具和技巧，这样你在计划自己的修改时，能够从计划如何测试它们开始。这正是测试驱动开发（test-driven development）技术的精髓：根据项目要求开发并实现必要的测试，编写代码，然后立刻执行这些测试。不熟悉这个概念的人可能会感到困惑：怎样才能为还没有编写出来的代码编写测试呢？接下来的几个小节将提供一些与这个越来越流行的概念有关的背景知识，希望它们能澄清大家心中的困惑。

4.1.1 为什么要测试

只要一提到关于软件质量的问题，就会有人问到这个问题。有些学生想知道进行多少测试才行。对于那些认为测试只是在浪费时间或吹毛求疵的学生，我会给他们一个机会让他们以测试工作量最小（有时甚至是没有）的策略来完成他们的软件工程的课程项目^①。

我的学生经常向我说起他们的模块和类编写得多么巧妙，他们是如何认真地遵守了建模原则。有许多学生还使用了UML（Unified Modeling Language，统一建模语言）图来帮助自己进行开发。这些做法都很好，但测试绝不仅仅是保证你的源代码与你的模型相匹配那么简单。有不少对自己的编程技巧很自信的学生交上来的作业经常存在着一些功能方面的小毛病。虽然那些小毛病一般不至于引发致命错误或崩溃（这在开发工作中经常会遇到），但在集成和软件的运行方面经常出问题。也就是说，这些学生没能做到让他们的软件按照顾客希望的那样工作。

如果你对这种场景很熟悉，就说明你已经了解了软件测试的价值。软件测试的形式有很多，目的都是保证和控制软件的质量。在什么时候选择什么技术进行测试是软件测试学需要解决的根本问题。

^① 这通常是一个持续整个学期的大型分组项目，从项目需求分析开始。

提示 如果你还没有过从测试入手进行软件开发或是与一位专业软件测试人员合作的经历,我建议你赶快去找一位这方面的专业人士。他们对软件的故障往往有着超常的洞察力,很少有开发人员能做到像他们一样。不要因为他们指出了你代码中的缺陷而感到羞愧——那是他们的工作,而且他们都精于此道!

1. 测试与调试

有不少人把“测试”和“调试”混为一谈。虽然它们的目标是同一个——找出缺陷,但它们并不是一回事。调试是一种交互式的过程,是通过分析源代码的内部工作情况去寻找源代码逻辑中的缺陷。测试是通过执行源代码去发现其中隐藏着的缺陷,不需要分析源代码的内部工作情况。

2. 测试驱动开发

测试驱动开发(test-driven development)通常与敏捷编程技术相关。事实上,测试驱动开发在采用极限编程(extreme programming, XP)方法的软件公司里最为常见。这听起来有点儿唬人,我可以向你透露一个关于XP的秘密:你不必采用XP方法也可以进行敏捷编程。

因为反对者不负责任的言辞,有不少人对敏捷编程技术持很深的怀疑态度。传统的软件工程方法学在理论和实践方面的成功是有目共睹的,人们把它奉为“金科玉律”也情有可原,但仅凭一知半解就把敏捷开发技术与偷工减料和质量低劣等同起来是毫无道理的,因为事实完全不是这样。

敏捷开发技术的核心概念是最大限度地减少软件开发过程中的不必要环节,重视与顾客的交流,只在需要的时候才编写需要的功能,把工作的重点放在满足客户此时此刻的需求上。敏捷开发技术围绕着顾客展开,具体过程并不是它最关心的焦点。敏捷开发技术可以全面推广,也可以有选择地在小范围应用。换句话说,在转向敏捷开发技术的时候,软件公司应该根据自己的具体情况摸着石头过河,而不是一下就扎进自己不熟悉的技术中,那会让它们的开发人员感到难以适应。欲速则不达,有不少软件公司就是因为在接纳敏捷开发技术方面太过急于求成才会陷入困境。正是因为这类负面案例时有发生,才会有人反对或怀疑敏捷开发技术^①。如果想了解关于敏捷开发技术与传统方法孰优孰劣之争的更多情况,请访问Agile Alliance网站www.agilealliance.org。

在各种敏捷开发技术当中,最有用的要属测试驱动开发技术了。测试驱动开发技术的基本思路很简单:从整个解决方案的一个基本模块开始,编写测试,运行测试,编写解决方案,用测试来验证它确实能够解决问题。这个过程听起来不难理解,但在实践中往往会变得相当复杂。在编写代码之前先编写测试让人有一种本末倒置的感觉——如果某个东西不存在,我们该如何去测试它呢?这么做真的有用吗?先行开发有关测试可以让你把精力集中到你的软件设计方案而不是代码上。我将解释一个典型的测试驱动开发过程,让大家看看测试驱动开发技术是如何紧扣设计方案并“驱动”程序员编写出源代码的。我知道这听起来有点儿奇怪,但我希望大家能给我个机会来证明它是有道理的。

测试驱动开发活动从整个系统的一个简单模块开始。用一个简单的类图把系统里的各个基本类以及它们之间的关系画出来。在这个类图中,除了一个暂定的类名外,各个类的代码块部分可以是空白。我说“暂定”是因为这通常是采用传统方法的程序员感到困惑的地方。在敏捷开发技术里,没有什么固定不变的,什么东西都允许修改。简单地说,只要能把顾客需要的软件交付给顾客,做什么都行。

^① 是的,采用敏捷开发技术的目的是为了减少不必要的工作,但如果转变得太快反而会把事情弄糟——我想说的就是这个意思。

设计好最初的类关系框图之后，把它复制下来放在旁边，它就是所谓的“域模型”(domain model)；它确定了整个系统中类的初始布局。接下来创建用例图(use case diagram)和辅助用例场景(用例及各种执行顺序的文本描述)。然后每个都用一个顺序图来扩充，这就制定出了所引用的类需要的函数。

等每一个类都大致成形之后，就可以开始编写测试了。是的，虽然那些类还都不存在，你仍然可以为它们编写测试。这些测试包括对域模型中的每一个类进行集成测试、系统测试和接口测试(它们都属于黑盒测试)。

注解 黑盒测试是在不了解系统内部构造的情况下进行的测试。白盒测试是在了解系统内部结构的情况下对其行为进行的测试。

就绝大多数以敏捷开发技术开发的项目而言，从这个过程的第一遍循环里学到的东西就是在这个时候融合到设计方案的相关部分(用例图、顺序图，等等)中并根据具体情况做出调整的。

注解 有些敏捷程序员还会使用健壮性图(robustness diagram)给这个过程增加一个建模步骤。这种做法与ICONIX过程很相似。关于ICONIX过程的更多信息请参考 *Agile Development with ICONIX Process*^① 一书。

这些修改包括：新类的发现、对现有类的重新组织以及定义某个类的方法和属性等。换句话说，在编写代码之前编写测试可以帮助你验证你的设计方案。这真的很酷：等你完成了你的设计并开始编写源代码的时候，你已经完成了测试！只需运行测试来证明你的代码的工作情况符合设计要求就行了。当然，如果需要改变测试或改变设计方案，你随时都可以这样做；这正是敏捷开发技术的魅力所在。

4.1.2 基准测试

基准测试(benchmarking)是一种测量和评估软件性能指标的活动。你可以在某个时候通过基准测试建立一个已知的性能水平(称为基准线)，当系统的软硬件环境发生变化之后再进行一次基准测试以确定那些变化对性能的影响。这是基准测试最常见的用途。其他用途包括测定某种负载水平下的性能极限、管理系统或环境的变化、发现可能导致性能问题的条件，等等。

基准测试的具体做法是：在系统上运行一系列测试程序并把性能计数器的结果保存起来。这些结构称为“性能指标”。性能指标通常都保存或归档，并在系统环境的描述中进行注解。比如说，有经验的数据库专业人员会把基准测试的结果以及当时的系统配置和环境一起存入他们的档案。这可以让他们对系统过去和现在的性能表现进行对照比较，确认系统或环境的所有变化。

基准测试通常都是些功能测试，即测试系统的某个功能是否达到了预期的要求。有些性能测试工具可以对系统几乎所有的方面(从最常见的操作到最复杂的操作，从小负载到中等负载到大负载)进行测试。

大部分程序员只在系统发生了奇怪的事情时才考虑进行基准测试，但我认为定期进行基准测试，尤其是在重大事件(比如系统或环境发生变化)之前和之后进行基准测试更有意义。一定要首先进行一次基准测试以创建基准线。如果没有基准线作为参照物，在事件发生之后进行的基准测试是不会对

① D. Rosenberg, M. Stephens, M. Collins-Cope. *Agile Development with ICONIX Process* (Berkeley, CA: Apress, 2005).

你有多大帮助的。

1. 优秀基准测试的指导原则

在进行基准测试的时候，有许多好的实践方法。在这一节里，我将向大家介绍几个我认为对大家最有帮助的基准测试原则。

首先，应该牢记“事前快照”和“事后快照”的概念。不要等到你对服务器做出修改之后才想起应该进行一次基准测试并把测试结果与你在六个月前建立的基准线进行对比。六个月的时间会发生许多事情！你应该在做出修改之前进行一次测试，做出修改，然后再对系统进行一次基准测试。这可以让你对三组性能指标进行对比：系统的预期性能、它在修改前的实测性能以及它在修改后的实测性能。你可以发现所发生的事情让你的改变多少会明显一些。比如说，假设你的基准测试有一项是度量查询时间。你在六个月前为某个特定的测试查询建立的基准线需要花费4.25秒才能完成。现在，你决定修改受测表的某个索引。你在修改之前进行的基准测试得到的结果是15.5秒，而你在修改之后进行的基准测试得到的结果是4.5秒。如果你没有拍摄事前快照，就不会知道你的修改让系统的性能有了很大的提高。说不定还会以为你的修改降低了查询的速度——你也许会因此撤消这次修改，结果返回到执行速度慢的查询。

虽然这是一个假想的例子，但我希望大家能够从中注意到以下几点。首先，如果你是在对某个系统的数据检索性能执行基准测试，而这个系统的数据量会随着时间的推移而增长，你必须更频繁地运行你的基准测试工具才能准确地把握数据量的增长对系统性能的影响。在刚才的例子里，你应该把有关性能指标（比如数据负载量）在事前的测量值当作系统的“正常”指标。

其次，必须保证你的测试对你测量的东西有效。如果你在对某个表的查询性能进行基准测试，你得到的测试结果只限于应用程序级别，不足以从一般意义上预测系统的性能。一定要把应用程序级别的基准与全局性的性能指标区分开来，这样才能保证不会得出错误的结论。

另外一个与事前概念和事后概念有关的好的实践方法是，在活动（负载量相对稳定）的有限时间内尽可能多做几次基准测试，这是为了保证你的测试结果不会受到局部活动（比如临时出现的进程或高资源占用任务）的影响。我发现重复进行几十次同样的基准测试可以把各次测试结果的平均值作为最终的性能指标值。有许多技巧可以得到这些统计结果。有条件的话，你甚至可以使用一个统计包或是你喜欢的适用于统计的电子表格应用程序^①来得出基本的统计数字。

注解 有些基准测试工具有自己的统计分析包，但MySQL Benchmark Suite没有。

我认为最有用的建议是每次只修改一个地方。一次修改多个地方并不是不可以，但这样你就不能期望从基准测试结果里得出什么有意义的结论。经常会发生这样的事：你修改了6个地方，其中之一产生的负面影响掩盖了另外几个的正面效果，剩下的一两个对性能没有任何影响。只有每次修改一个地方，你才能准确地判断出它对系统性能的影响是负面的、正面的还是没有影响。

还有，只要有可能，就应该使用实际数据来进行基准测试。人工生成的测试数据怎么说也会有一些规律可循，那样得到的测试结果往往不能反映实际情况，某些特定的功能（比如边界值和范围检查等）可能永远也得不到测试。如果你的数据变化很频繁，你应该选择某个时刻为它们“拍摄”一张快

^① 有些统计专家认为Microsoft Excel的统计引擎不够精确。但我认为，就你们想测量的数据而言，精确度不够并不是什么大问题。

照，然后使用这张快照来进行每一次测试。不过，这么做虽然能够保证使用真实的数据来测试性能，可是随着数据量的增长也许无法测试出系统性能的下降。

最后，在解读基准测试结果和管理预期目标时，一定要让你的目标有现实意义。如果你想改善系统在某种特定条件下的性能，在确定目标前首先要把已知的后果弄清楚。比如说，如果你想知道把网络接口的传输速度提高100倍对系统性能会产生哪些影响，就必须先弄清楚你的服务器将不能按照比现在快100倍的速度发送和接收数据。在这类场合中，你必须综合考虑硬件的性价比和硬件可能带来的性能改善。换句话说，你的服务器的执行速度应当提高几个百分点，这样就为你省了钱（或说增加了收入）。

如果在做过仔细评估之后预计你的网络性能只要提高10%就可以做到收支平衡甚至赢利，那就把这个数字作为你的目标好了。如果基准测试结果表明你得到了这么大（或更好）的改善，就去找老板谈谈加薪的事吧；如果基准测试结果表明你没有得到这么大的改善，去建议老板把新硬件退回去（也可以顺便谈谈加薪的事，因为你让他省钱了）。不管是哪种情况，你的报告都有充分的依据，即你的基准测试结果。

2. 对数据库系统进行基准测试

基准测试在很多领域都非常重要。但基准测试与数据库服务器到底有什么关系呢？答案包括很多方面。

对数据库服务器进行基准测试可以在许多不同的层次上进行。最常见的是针对数据库模式的改动而进行的基准测试。专门针对某个表的基准测试比较少见（虽然你可以这么做）。人们更感兴趣的是在改变了数据库的结构之后，其性能会受到什么样的影响。

人们的这种关心在刚开始使用一个新的应用程序或一个新的数据库时表现得尤为明显。此时，你可以设计好几种数据库模式并填充数据，然后编写一些基准测试程序来模仿所推荐的系统。嘿，这也是一种测试驱动的开发行为！通过创建多个数据库模式并进行基准测试，甚至可能会多次重复这些改动，你很快就可以确定哪套模式最适合你设计的应用。

有时候，对数据库系统进行基准测试还有一些特殊的目的。比如说，你想知道数据库系统在不同的负载情况或不同的系统环境下会有怎样的性能表现。那么，除了进行事前和事后的基准测试去了解对环境所做的改变会产生多大的不同，还有什么方法更能证明你新安装的RAID设备将大幅改善系统的性能呢？是的，一切都是围绕成本进行考虑，基准测试工具可以帮助你管理好数据库系统的成本。

4.1.3 性能分析

有些故障现象只在系统的负载量达到一定限度之后才会出现。这类问题通常表现为系统变慢，但不会出错。怎样查找这类问题呢？你需要一种能在系统保持运转的同时检查其工作情况的方法。这个过程就是所谓的性能分析（profiling）。有些作者把性能分析和调试放在一起讨论，我也承认性能分析是一种调试手段，但我认为性能分析远不止是一种调试工具。性能分析可以让你在基准测试之前就能发现性能瓶颈和潜在隐患。可是性能分析往往是在发现问题苗头之后，且有时作为一种追查其根源的手段来进行的。通过性能分析可以发现或性能分析的问题包括内存和磁盘的消耗量、CPU占用量、I/O应用、系统响应时间以及其他一些系统参数。

有不少人把性能分析（或性能分析器）与对某些个系统参数进行测量混为一谈。用术语来说，测

量某个性能指标称为诊断操作或诊断技术（有时也叫作“跟踪”），负责管理这些诊断操作并对有关系统进行这些诊断的系统叫作性能分析器。因此，性能分析就是应用性能分析器来完成各种诊断操作。

绝大多数性能分析器都可以把系统在某特定时间段里的运转情况生成为一份机读报告。这类报告里的性能测量值通常称为踪迹，因为它们一直在跟踪系统的运行情况。有些性能分析器也可以生成适合人类阅读的报告，它们可以把系统在哪些地方耗时最多的情况详细列出来。这类性能分析器通常用来性能分析各种系统级资源，如I/O、内存、CPU、线程或进程等。比如说，你可以发现线程和进程正在执行什么命令或函数。如果你的系统还在线程或进程的首部里记录着其他的元数据，你也许还能发现与线程或进程的阻断和死锁有关的性能问题。

注解 举个例子，当两个进程各自锁定了一项资源、同时又都在等待对方先释放它锁定的资源才能继续执行时，就产生了死锁。能否有效地监测死锁，是评价一个数据库系统是否设计优良的重要标准。

你还可以通过性能分析找出性能最差的查询，甚至可以追查出哪些线程或进程的执行用时最长。在这类情况下，你也许还能发现某个特定的线程或进程正在消耗大量的资源（如CPU或内存等）并采取相应的步骤去纠正错误。在一个有大量用户同时访问某些中央资源的环境里，这种情况并不少见。

有时候，某些特定的系统请求可能导致某一个用户的操作（合法的或不合法的——我们希望它是合法的）会影响到其他用户的情况。此时，你可以正确地找出那个引起麻烦的线程或进程和它的属主，然后迅速采取步骤消除隐患。

在开发一个系统的过程中，性能分析还可以作为一种强大的诊断工具来使用，这也正是人们想要把它们归入调试工具的原因。系统报告的类型可以帮助你发现在你的源代码里都存在着哪些可能导致性能低劣的缺陷。不过，千万注意不要做过了头——有些源代码的性能分析确实需要较长的时间才能执行完毕，花费大量的时间去对这些源代码进行性能分析不一定有助于发现真正的瓶颈。请记住，任何一种操作都需要花费一定的时间才能完成，磁盘I/O操作和网络通信中的合理延迟就是如此。在很多时候，如果你想让系统摆脱对某个慢速资源的依赖，唯一的办法就是重新设计它的体系结构。当然，如果你正在开发的是一个嵌入式实时系统，重新设计其体系结构确实值得一试，但如果事情根本不在你的控制范围内，花费大量的时间和精力去改善性能往往不会带来什么实际的收获。

话虽如此，你还是应该让代码有着尽可能高的执行效率。如果通过性能分析发现代码还有改进的余地，就应该立刻进行改进。在大的问题解决后，一定要继续查明或跟踪小的问题。

基准测试与性能分析

基准测试与性能分析之间的差异有时很模糊。基准测试用于建立性能评估或测量，性能分析则用于从性能方面识别系统行为。

基准测试用于建立系统在特定配置下已知的性能特征，而性能分析可以让你知道系统在哪些环节执行的时间最长。进行基准测试的主要目的是为了系统能够按照或者达到一个给定的标准（基准线）运行得更好，进行性能分析的主要目的则是为了找出系统中的性能瓶颈。

4.1.4 软件测试简介

软件测试技术是计算机科学里的一个研究领域。这个领域对IT行业有着越来越重要的意义，因为长期以来的实践早已证明软件系统的故障大多是因为没有进行足够的测试或没有足够的时间进行测试而造成的。

但是测试手段和测试目的本身却经常成为人们争论的焦点。比如说，有些人认为进行测试的目的是为了发现缺陷。这听起来很合乎逻辑，不是吗？但仔细想想，这种说法隐含的意思是只有发现了缺陷的测试才是成功的！那么，那些没有发现缺陷的测试算不算成功呢？是因为测试本身存在着问题，还是因为被测试的东西真的没有任何缺陷？直到现在，软件测试研究人员还在为这些话题（以及许多其他事情）争论不休。

有些软件测试人员（以下简称为“测试员”）认为没有发现任何缺陷的测试才是成功的，这句话里隐含的意思与上面的说法——“只有发现了缺陷的测试才是成功的”，是不一样的。如果你站在这些测试员的立场上看，即使某个系统已经通过了所有的测试（所有的测试都成功了），它仍有可能存在着缺陷。在这种情况下，焦点是接受测试的软件而不是测试本身，即使在测试之后又发现了缺陷，也很少认为是测试的失败。但是，如果从“只有发现了缺陷的测试才是成功的”角度看，只有当接受测试的软件没有任何缺陷时测试才会失败。因此，如果没有找到缺陷，就说明测试本身还需要改进。我们从两方面来看待这个问题是有原因的。

4.1.5 功能测试与缺陷测试

测试员的主要工作是确保系统达到了说明书（也叫需求文档）的要求。通常，测试的目的是为了证明系统能够提供用户所需要的功能而不是寻找缺陷。这类测试通常被称为功能测试（functional testing）或系统测试（system testing）。功能测试不关心系统的内部工作情况（黑盒测试），测试者通常是从一名最终用户的角度对有关软件进行一系列操作。比如说，假设某个系统提供了一项打印功能，相应的功能测试将模拟各种可能的用户环境和用户操作情况对这项打印功能的各个方面进行检查。此时，如果打印过程没有出错，打印结果也没有问题，测试就是成功的。功能测试只是软件工程师和测试员可以用来保证产品质量的众多测试手段中的一种。

而缺陷测试的目的则是输入一组有效的或无效的数据来使系统出错。缺陷测试往往需要了解软件的内部工作原理（通常称为白盒测试），然后有针对性地使用一系列合法或非法的输入数据去“诱惑”系统出现故障。这意味着测试员必须先把被测系统的某个组件的源代码分析透彻，列举出所有可能的执行场景（或路径），然后编写一系列测试去检查每一条执行路径上的每一个临界值和阈值。比如说，还是刚才那个打印功能的例子，相应的缺陷测试不仅要对该功能的所有正常操作进行测试，还要对该功能的出错处理器（error handler）和异常触发器（exception trigger）做全面的测试。也就是说这些测试是故意让代码出错，而没能发现任何缺陷的测试将被认为是失败的。所以现在人们经常用“正”（到达测试目的）和“负”（没有达到测试目的）来描述测试结论。^①

本书将把功能测试和缺陷测试结合在一起进行，即在进行功能测试的同时对其内建的功能进行缺陷测试。我们使用的测试机制能够让大家通过执行一系列SQL语句来测试MySQL服务器的功能，并在

^① 关于软件测试技术的更多信息可以在http://en.wikipedia.org/wiki/Software_testing主页上找到。

此基础上对MySQL服务器进行缺陷测试。事实上，我建议所有的测试都用来对相关的出错处理和异常处理功能进行测试。如果你的测试没有发现缺陷或者事后有人向你报告了一个缺陷，你应该赶快编写一个新的测试或修改一个现有的测试去查找它的根源。这样，你就可以肯定：在修订这个缺陷之前，它能够重复发生，之后会标明该缺陷已经修复。

1. 软件测试的类型

一般来说，软件测试都是在一定的约束条件下进行的。它们的第一步通常都是分析受测系统的项目要求和设计方案，然后根据项目要求和设计方案编写一系列测试程序去评估软件的质量（正确性、健壮性和可用性等）。正像我前面提到的那样，有些测试是为了查找缺陷，有些测试是为了验证功能正常（请注意，“功能正常”不等于“没有缺陷”）。还有一些测试是为了对系统做出某种评估或认证，这类测试大都偏重于质量因素而不是数字结果。

软件测试技术是软件工程领域的一个组成部分，其目的是为了保证软件能够达到设计要求并具备人们所预期的功能。这个过程有时被称为验证（validation）和检验（verification）。这两个词很容易混为一谈。“验证”只是检查软件是否达到了项目要求。“检验”是检查软件的开发过程是不是按照正确的流程和方法来进行的。换句话说，“验证”回答的问题是“我们的产品有没有问题”；“检验”回答的问题是“我们开发这个产品的过程有没有问题”。

许多软件开发流程都有检验和验证环节，但绝大多数程序员都习惯于把评估项目要求是否得到满足的验证环节称为软件测试。此外，验证环节的主要任务是测试某个系统的各项功能是否齐备并且不存在功能方面的缺陷，而不是查找和纠正软件里的错误。

对软件进行测试的具体做法有许多种。事实上，在项目计划的早期，人们经常会因为应该进行哪些测试不应该进行哪些测试而争论不休。当然，绝大多数程序员都同意测试环节是软件开发工作的重要组成部分。不过，就我个人的经验而言，知道不同的软件测试技术各自适用于何种场合的人并不多。只有你才能为你的项目做出正确的选择，而我的目标是把一些比较流行的软件测试技术介绍给大家，希望能帮助你根据自己的具体需要选出最适用的测试来进行。

在接下来的几个小节里，我将介绍几种比较流行的软件测试技术、其目标和适用范围、它们与测试驱动开发之间的关系。你将看到，软件测试其实是一个有始无终的过程，传统意义上的阶段性测试是这个过程中的里程碑。

● 集成测试

集成测试（integration testing）发生在把系统的基本构造块集成为一个整体的时候。集成测试先从某一个简单的组件开始进行测试，然后把测试范围逐步扩大到其他的组件，直到整个系统被集成为一个整体为止。这种测试在各组件既相对独立、又彼此相关的大型开发项目里最为常见。

● 组件测试

组件测试（component testing）指的是对系统的某个半独立部分（组件）单独进行的测试。简单地说，就是通过依次调用某组件里的每一个方法、依次读/写/使用该组件里的每一个属性来进行测试。组件测试通常需要编写一些比较大的测试程序来提供测试组件所需要用到的一切外部通信，与被测组件有关系的所有其他组件需要用代码脚手架（有时称为模拟组件或存根组件）模拟出来。这些代码脚手架提供了所有通信必需的输入和输出，并对被测组件进行试验。

● 接口测试

接口测试（interface testing）针对的是组件的接口而不是组件本身，其目的是检查某个组件的接

口是否具备它必须具备的全部功能。这类测试通常与组件测试一起进行。

● 回归测试

回归测试 (regression testing) 的目的是为了保证对被测软件的任何增加或修改不会影响到该软件的其他部分。稍微具体地说,就是把以前运行过的测试再全部运行一遍,然后把这次测试结果与以前的结果进行比较。如果结果相同,就说明新改动的部分不会影响其他的功能(当然,这还要看测试程序是如何编写的)。这类测试通常是用某种能够在无人值守的情况下自动进行有关测试的自动化测试软件来完成的,它们可以在完成一组测试之后对前后两次的结果进行对比。自动测试是敏捷开发技术中的一个热门概念。

● 路径测试

路径测试 (path testing) 说的是采用穷举法对每一条可能的执行路径进行测试。这种测试需要测试者对被测试的源代码有全面的了解(白盒测试),其目的通常不是为了检查某个软件是否达到了项目要求,而是为了检查它能否准确无误地沿着每一条可能的路径从开始执行到最后。路径测试通常与功能测试一起进行。

● α 阶段测试

传统意义上的 α 阶段测试 (alpha stage testing) 是在项目开发工作进行到了一定阶段、软件的质量已经足够稳定的时候开始的。 α 阶段测试一般是在软件已经能够基本满足日常应用的需要的早期进行的。这个阶段的测试有时是为了证明被测软件已具备足够的稳定性,绝大多数功能都已可以交付使用(也许还有一些细小的缺陷)。这可能包括运行一些用于验证系统在保护状态下运行情况的部分测试。通常,通过 α 阶段测试之后,软件的开发工作就基本告一段落了,除一些已知的小的和中等的缺陷外,不会有大的缺陷。在软件通过 α 阶段测试之后,开发工作的 α 阶段也就结束了,接下来将进入项目的 β 阶段。

测试驱动开发技术是这样看待 α 阶段测试的:系统已基本完成,所有测试都将在真实的代码上进行,不再需要使用脚手架(存根类)。一旦测试结果表明开发工作可以进入 β 阶段,项目就将进入到它的 β 阶段。

● β 阶段测试

按照人们的普遍理解,当某个新系统的全套功能都已实现出来并通过 α 阶段测试,只有少数功能还需要进一步提高效率或加强(加固)的时候,该系统的质量就将被认为是稳定的,可以用于日常工作。在这个阶段运行的测试通常是涉及所有功能的全套测试;即使发现问题,往往也只是一些小毛病。这类测试通常还会邀请项目的委托方/顾客或是一些潜在的用户来参与。虽然这些人所使用的测试方法可能不那么科学,但他们为开发者提供了一个让其产品接受实践检验的机会。就算没有发现什么大的问题,开发者也可以根据用户的反馈意见对其产品做一些小修改,让它更加完善,更受欢迎。通过了 β 测试意味着软件本身已随时可以正式发布。

在测试驱动开发技术里, β 测试是持续开发测试过程中的又一个里程碑。测试驱动开发技术是这样看待 β 阶段测试的:根据测试结果,绝大多数功能都性能良好,系统的稳定性(通常以在测试中发现的缺陷的个数来衡量)已达到可以交付使用的程度。

● 发布测试、功能测试和验收测试

发布测试 (release testing) 通常是一些验证系统达到项目要求的功能测试,在软件交付之前进行,类似于 β 阶段,有些软件公司也会邀请顾客参与这类测试。因此,这类测试往往也被称为验收测试

(acceptance testing),意思是由用户来评判被测软件是否达到了他们的要求。测试驱动开发技术把这些里程碑当作测试过程的终点。

● 易用性测试

易用性测试(usability testing)通常是在系统接近完成或者是已经完成时进行的,它们往往与功能测试和发行测试同时进行。进行易用性测试的目的是为了评估用户在与系统打交道时的体验是否良好。这类测试的结果通常不是“成功”或“失败”,而是一系列“喜欢”和“不喜欢”。以用户的个人喜好作为主要评判标准虽然不够客观,但易用性测试可以帮助开发者推出更受欢迎的软件和赢得用户的忠诚度。易用性测试有时在专业的实验室里进行,那里有专人或设备把用户的反应和建议记录下来供稍后分析。不过,绝大多数易用性测试都不那么正规,比较常见的做法有两种:一是开发者征召一些用户来观察他们使用新软件的情况;二是让一些用户在一段时间内免费使用新软件,然后填写一份调查问卷或面谈。

● 可靠性测试

可靠性测试(reliability testing)考察的是系统在不同负载水平下保持长时间正常运转的能力,这里所说的负载有质量(数据的复杂程度)和数量(数据量的大小)两重含义。衡量可靠性的标准主要有两个:一是系统持续运行的时间,二是每个小时或每次测试所监测到的缺陷的个数。

● 性能测试

这里说的性能测试(performance testing)包括两个含义:一是了解系统的性能指标(基准测试),二是考察系统的实际性能是否与预期目标相符。这种测试往往同时兼顾可靠性和性能两个方面。有时候,系统在极限负载下的性能(术语称之为“强度测试”)就是在这类测试期间被检验的。

注解 易用性测试、可靠性测试和性能测试既是传统意义上的测试,也可以在测试驱动开发环境里随时进行。

2. 测试设计

在概括地介绍了软件测试技术和一些常见的测试类型之后,我们再来讨论一下如何制定测试方案的问题。可以用来制定测试方案的思路有很多,它们的最终目标不外乎测试、验证或审查某个软件的特定方面或它的开发过程。下面是3种最基本的测试方案设计思路。

● 基于规格说明的测试

这种类型的测试(有时称为功能测试)是测试软件的需求和设计的。主要是验证软件是否达到了项目要求。通常根据一个或一组给定的项目需求去来进行,测试步骤和测试程序按功能划分为一系列集合(有时称为测试集)。在系统的开发过程中,测试集可以在某项功能刚被实现出来时用来检查它是否满足有关要求(功能测试),还可以在此后随时用来检查它是否仍然满足有关要求(也叫回归测试)。

● 划分测试

划分测试(partition testing)的重点是检查某个系统的输入和输出特性。典型的做法是根据输入或输出的取值范围选用三种数据进行测试,即界外、边界和临界值。比如说,假设某个系统的输入是1到10之间的一个正整数,我们就可以通过测试{0, 1, 5, 10, 11}中的值来对这一数据进行分区(称为等价性划分或等价性域)。有些软件测试工程师还会再加上一些负数(比如-1)来测试它。这里的基本思路是:如果被测系统执行了范围检查,边界条件往往会比有效数据甚至无效数据更可能揭示出系统中的缺陷。

在我们刚才提到的例子里，如果你想测试内部数据收集代码（用来读取并解释输入值的那个部分）的代码，就没有必要测试比11还大的整数。事实上，因为现如今的绝大多数系统（比如Microsoft Windows Forms）都使用系统级调用来管理数据的输入，其本质上是非常可靠的，所以我们通常不需要对数据收集部分进行测试。最有意思的是你还可以按照上述思路为输出数据构造一个测试集，此时的测试目的将是系统如何根据给定的输入数据（好或坏）产生输出（看它是好还是坏）。用术语来说，就是检查该系统的健壮性和它在处理输入数据时的精确度。划分测试在验证某个系统的性能和健壮性方面非常有用。

● 结构测试

按照这一思路设计出来的测试方案通常被称为结构测试（structural testing）或体系结构测试（architectural testing），其重点是检查系统的布局（或体系结构）是不是符合有关要求，即检查该系统的结构是否符合预期。这类测试通常分两个步骤进行：先检查各组件本身的接口是否完备，再检查各组件之间能否正确协作。这些测试类别涵盖了所有白盒测试的方法，其目的是检查系统中的每一条执行路径是否都能正确地执行到底（路径测试）。这些测试被看作是不同的验证，因为它们证实了体系结构是否正确构建，是否遵循了预定的流程。

4.2 MySQL 测试

测试MySQL系统的办法有很多。你可以用mysqlshow命令来测试服务器的连接性和基本功能，可以使用各种客户工具手动地运行各种测试，可以使用基准测试工具来测评其性能特征，甚至还可以对服务器进行性能分析。绝大多数数据库专业人员首选的工具是MySQL Test Suite和MySQL Benchmarking Suite。以下几个小节将介绍这两种工具及其使用技巧。

4.2.1 MySQL Test Suite

MySQL Test Suite是MySQL AB公司面向大众推出的一套功能强大的测试工具。这套测试软件包括一个名为mysqltest的可执行文件（Windows版本是mysql-test.exe文件）和一系列用来测试系统和对比测试结果的Perl模块和脚本。表4-1列出了相关目录及其内容。这套测试软件有适用于Unix/Linux平台的二进制和源代码发行版本，部分适用于Windows平台的发行版本也包括它。

注解 MySQL Test Suite工具目前还不能直接在Windows环境下运行。如果你想通过MySQL代码贡献项目为MySQL开发做贡献，这会是一个很不错的项目。在Windows平台上，如果已经安装了Perl环境和Perl DBI模块，这个工具可以在Cygwin环境里运行。详见《MySQL参考手册》中的2.13节。

表4-1 mysql-test目录下的子目录

子 目 录	说 明
/misc	一些辅助性的Perl脚本
/ndb	全套的集簇测试
/r	测试运行后的结果文件
/std_data	这套测试软件所使用的测试数据
/t	各种测试

在安装好MySQL之后，你可以在其安装目录下的mysql-test子目录里找到一个名为mysql-test-run.pl的Perl脚本。这套测试软件的最大优点是易于扩充，你可以把自己为某个特定的应用软件或需要而编写测试程序进行测试。测试可用来进行回归测试，运行该测试来检查所有的功能是不是仍像以前那样工作。

所有的测试程序都集中保存在子目录mysql-test的一个名为/t的下级子目录里。这个子目录包含着将近600个测试。这听起来已经是个不小的数字了，但MySQL的随机文档称这些测试仍无法涵盖MySQL系统的全部功能。这套测试工具的最新版本可以帮助我们查出绝大多数SQL命令、操作系统和库交互、以及MySQL群集和复制功能里的缺陷。MySQL AB公司一直希望这个工具包能收集足够多的测试以涵盖整个MySQL系统。事实上，MySQL AB公司一直在公开收集着各种测试，其目标是推出一个能够对MySQL系统所提供的100%的功能进行各种必要测试的工具包。如果你觉得自己编写的测试程序弥补了mysql-test/t子目录里的某个现有测试的不足，请赶快把你的测试提交给MySQL AB公司。

提示 你可以从MySQL Internals邮件列表上查到更多关于MySQL Test Suite的信息（详见<http://lists.mysql.com/>；那里还可以查到可用的列表），你也可以把你的测试通过电子邮件提交给这个邮件列表，但MySQL AB公司推荐的办法是把你的测试文件上传到<ftp://ftp.mysql.com/pub/mysql/upload>站点。这里要提醒大家一句：如果你想把测试提交给MySQL AB公司并希望它被收录到这个工具里，你使用的测试数据任何人都可以查看。要知道，这个工具包的源代码对全世界的每一个人都是开放的。比如说，你的亲朋好友肯定不希望他们的电话号码出现在每一台装有MySQL系统的机器里。

对于每一项测试，相应的结果文件都保存在/mysql-test/r子目录里。那些结果文件的内容包含着相关测试程序的输出，这些输出将与事先保存的“标准”结果进行比较（用diff命令）。在很多时候，结果文件相当于测试程序所输出的基准。你可以创建一个测试并保存其运行的结果，等以后再运行这个测试时，就可以通过对比这两次的测试结果而知道你的系统是不是和以前一样了。

但是，使用这个办法要小心。有些数据时时都在变化，这些数据可以用在测试里，但必须使用附加的命令才能正确地处理好它们。可惜的是，MySQL Test Suite默认忽略像这样的数据而不是直接进行比较。因此，如果在测试中使用了时间和日期字段这样的数据类型，就可能会引起一些麻烦。我将在稍后再讨论这个问题和其他命令。

1. 运行测试

用这个测试工具来运行测试很容易，进入mysql-test子目录并执行./mysql-test-run.pl命令即可。这将启动这个测试工具的主控程序，连接到服务器，然后运行/t子目录里的所有测试。什么，你不想运行全部600个测试？因为一次运行这么多的测试需要花费不少时间，所以MySQL AB在编写这个测试工具的时候留了一手，让人们可以有选择地一次只运行几个测试。比如说，下面这条命令将只运行名为t1、t2和t3的测试：

```
%> ./mysql-test-run.pl t1 t2 t3
```

这个测试工具将依次运行每一个测试，但一旦某个测试失败了，这次测试就会半途中止，不再继续运行下一个测试。如果你不想让你的测试半途而废，就要在命令行上明确地给出--force参数。

在默认情况下，这个测试工具会启动一个它自己的mysqld实例，这有可能与正在你机器里运行着

的另一个实例发生冲突。因此，在运行这个测试工具之前，你应该先关闭MySQL服务器的其他实例。如果你是从源代码目录里使用这个测试工具的，可以通过编译源代码的办法来创建你自己的mysqld可执行文件；这种安排在你想对修改过的服务器进行某种测试，但又不想或不能把你当前使用的服务器替换下来的时候非常方便。

注意 只要当前使用的服务器没有用到3306或3307号端口，你就可以在不关闭它的情况下运行这个测试工具。如果不是这样，这个测试工具有可能运行不正确。你应该关闭当前服务器或者改用其他的端口。

如果你想连接到某个特定的服务器实例，可以使用--extern命令行参数来告诉测试工具连接服务器。如果你有额外的启动命令或是想使用某个特定的用户去连接服务器，同样可以加上相应的命令。如果想了解更多关于mysql-test-run脚本的命令行参数的信息，请输入下面这条命令：

```
%> ./mysql-test-run.pl --help
```

或者，也可以到<http://dev.mysql.com/doc/mysql/en/mysql-test-suite.html>了解更多详细说明。

注解 如果使用了--extern命令行参数，就必须把你想要执行的测试的名字也写出来。有些测试需要一个本地的服务器实例才能执行。比如说，下面这条命令将连接到一个正在运行的服务器并执行alias和analyze测试：`%> perl mysql-test-run.pl --extern alias analyze`

2. 创建新测试

如果想创建一个测试，可以用一个标准的文本编辑器在/t子目录里创建一个名为mytestname.test的文件（文件名自己选，后缀必须是test）并把你的测试代码写在该文件里即可。比如说，下面是我创建的一个名为cab.test的示例测试（见代码清单4-1）。

代码清单4-1 示例测试

```
#
# Sample test to demonstrate MySQL Test Suite
#
--disable_warnings
SHOW DATABASES;
--enable_warnings
CREATE TABLE characters (ID INTEGER PRIMARY KEY,
                          LastName varchar(40),
                          FirstName varchar(20),
                          Gender varchar(2)) TYPE = MYISAM;
EXPLAIN characters;
#
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (3, 'Flintstone', 'Fred', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (5, 'Rubble', 'Barney', 'M');
```

```
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (7, 'Flintstone', 'Wilma', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (9, 'Flintstone', 'Dino', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (4, 'Flintstone', 'Pebbles', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (1, 'Rubble', 'Betty', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (6, 'Rubble', 'Bam-Bam', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
    VALUES (8, 'Jetson', 'George', 'M');

#
SELECT * FROM characters;
#
EXPLAIN (SELECT DISTINCT LASTNAME from characters);
#
SELECT DISTINCT LASTNAME from characters;
#
# Cleanup
#
DROP TABLE characters;
# ...and we're done.
```

这个示例测试的内容都是些简单的SQL命令，创建了一个数据表，插入了一些数据，然后又进行了几个简单的查询。绝大多数测试都会比这个例子复杂，但道理是一样的。创建测试来运行一些命令（或进行一些数据处理）。请大家注意前6行。前3行是以#字符开头的注释行。在创建你自己的测试时，也应该像这样在文件的开头部分简单地解释一下这个测试到底要干什么。推而广之，只要测试里有一些不容易理解的命令（比如复杂的关联操作或用户定义函数），就应该在代码中间对它们做出必要的解释。第4行和第5行很有意思，因为它们是在向这个测试工具发出命令。这个测试工具的命令总是以--开头。这两行代码是在告诉这个测试工具临时禁用，然后再重新激活来自服务器的警告消息。这很有必要，因为那个表（characters）此时尚不存在。如果不禁用这些警告消息，这个测试在遇到以下两种情况时就会半途而废：

- ❑ 从服务器返回了一条警告消息。
- ❑ 输出与预期结果不匹配。

作为一条基本原则，在测试的开头应该先对上次测试失败遗留下来的临时表或视图进行一下清理。正式的测试部分应该包括完成这个测试所需要的所有语句。最后，在测试结束的时候，还应该对本次测试可能遗留下来的表或视图再做一次清理。

提示 在编写测试时，MySQL AB公司要求用像t1、t2、t3这样的名称来命名有关的表，像v1、v2、v3这样的名称来命名视图；这是为了保证你的测试数据表不会与现有的任何测试表发生冲突。

3. 运行新测试

把测试程序编写好以后，需要执行测试并创建预期结果的基准线。从mysql-test子目录执行以下命令来运行新创建的cab.test测试：


```
%> touch r/cab.result
%> ./mysql-test-run.pl cab
%> cp r/cab.reject r/cab.result
%> ./mysql-test-run.pl cab
```

第一条命令将创建一个空的结果文件。这是为了保证这个测试工具有东西可以比较而必需的。下一条命令是第一次运行这个测试。代码清单4-2给出了一个典型的首次运行的测试结果。请注意，这个测试工具报告显示本次测试失败了，这是因为此时还没有可供比较的结果。为简明起见，我省略了一些不太重要的语句。

代码清单4-2 首次运行一个新的测试（简略版）

Starting Tests

TEST	RESULT
cab	[fail]

Errors are (from /home/Chuck/MySQL/mysql-5.1.9-beta/mysql-test/var/log/mysqltest-time) :

mysqltest: Result length mismatch
(the last lines may be the most important ones)
Below are the diffs between actual and expected results:

```
*** r/cab.result      2006-05-24 03:40:46.000000000 +0300
--- r/cab.reject      2006-05-24 03:42:50.000000000 +0300
*****
```

Ending Tests

Shutting-down MySQL daemon

Master shutdown finished
Slave shutdown finished

Failed 1/1 tests, 00.0% were successful.

接下来的命令把最新的结果从cab.reject文件里复制到cab.result文件。这个步骤只需要做一次，即你能肯定你的测试可以正确无误地运行。要想做到这一点，最保险的办法是先以手动方式逐条执行你的测试语句并确认它们的执行情况都没有问题，然后再把*.reject文件复制成一个*.result文件。代码清单4-3是这个新测试最终的结果文件，它的内容与你手动执行那些测试语句时看到的输出是一样的，只是排版效果和数据列之间的间隔稍微逊色了一些。

代码清单4-3 结果文件

```
DROP TABLE if exists characters;
CREATE TABLE characters (ID INTEGER PRIMARY KEY,
```

```

LastName varchar(40),
FirstName varchar(20),
Gender varchar(2));
EXPLAIN characters;
Field Type Null Key Default Extra
ID int(11) NO PRI
LastName varchar(40) YES NULL
FirstName varchar(20) YES NULL
Gender varchar(2) YES NULL
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (3, 'Flintstone', 'Fred', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (5, 'Rubble', 'Barney', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (7, 'Flintstone', 'Wilma', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (9, 'Flintstone', 'Dino', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (4, 'Flintstone', 'Pebbles', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (1, 'Rubble', 'Betty', 'F');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (6, 'Rubble', 'Bam-Bam', 'M');
INSERT INTO characters (ID, LastName, FirstName, Gender)
VALUES (8, 'Jetson', 'George', 'M');
SELECT * FROM characters;
ID LastName FirstName Gender
3 Flintstone Fred M
5 Rubble Barney M
7 Flintstone Wilma F
9 Flintstone Dino M
4 Flintstone Pebbles F
1 Rubble Betty F
6 Rubble Bam-Bam M
8 Jetson George M
EXPLAIN (SELECT DISTINCT LASTNAME from characters);
id select_type table type possible_keys key key_len ref rows Extra
1 SIMPLE characters ALL NULL NULL NULL NULL 8 Using temporary
SELECT DISTINCT LASTNAME from characters;
LASTNAME
Flintstone
Rubble
Jetson
DROP TABLE characters;

```

有了预期结果之后，我再次运行了这个测试。这一次，这个测试工具报告显示测试通过了。代码清单4-4给出了一个典型的测试结果。

代码清单4-4 一次成功的测试运行

```

Installing Test Databases
Removing Stale Files
Installing Master Databases
running ../sql/mysqld --no-defaults --bootstrap --skip-grant-tables
    --basedir=. --datadir=./var/master-data --skip-innodb
    --skip-ndbcluster --skip-bdb
    --language=../sql/share/english/
    --character-sets-dir=../sql/share/charsets/
Installing Slave Databases
running ../sql/mysqld --no-defaults --bootstrap --skip-grant-tables
    --basedir=. --datadir=./var/slave-data --skip-innodb
    --skip-ndbcluster --skip-bdb
    --language=../sql/share/english/
    --character-sets-dir=../sql/share/charsets/
Manager disabled, skipping manager start.
Loading Standard Test Databases
Starting Tests

TEST                                RESULT
-----
cab                                [ pass ]
-----

Ending Tests
Shutting-down MySQL daemon

Master shutdown finished
Slave shutdown finished
All 1 tests were successful.

```

创建你自己的测试并运行它们很容易做到。无论你想编写多少个测试，只要按照我刚才描述的步骤来进行——有些测试可能需要你多重复几遍这一过程，就应该不会有什么问题。可以看到，这个过程与测试驱动开发技术的理念是相吻合的：先创建测试，在没有结果证据的情况下运行它，创建解决方案（预期结果），然后再执行这个测试。如此反复直到测试取得成功为止。我建议大家在自己的MySQL应用软件（尤其是在扩展MySQL服务器）的时候也采用这样的套路。

比如说，假设你想创建一个新的SHOW命令。此时，你应该创建一个新的测试来执行这个新命令，运行它并建立测试。当然，在你真地创建出这条新命令之前，你的测试肯定每次都会失败。这个套路的好处是它可以让你动手编写代码之前，把注意力集中在命令的执行结果和命令语法上来。如果你决定采用这个套路来开发所有的项目，肯定不会后悔，因为你将在代码的质量方面得到很高的回报。一旦实现了这个命令并通过再次运行测试和检查*.reject文件（或以手动方式执行命令）的办法证明它没有问题，就可以把*.reject文件复制为*.result文件，测试工具将使用这个*.result文件去验证你下次执行这个测试的结果（通过/失败）。

4. 高级测试

MySQL Test Suite提供了丰富的命令用来创建强大的测试。本小节将对一些比较常用和有用的命令进行介绍。令人遗憾的是，MySQL的随机文档没有对这些命令进行解释，以下内容是我通过分析有关的测试程序和网上的帖子收集来的。

提示 如果你使用了MySQL Test Suite的高级测试命令，可以用--record命令行参数来创建结果文件以记录适当的结果。比如说，你可以用命令/mysql-test-run.pl --record cab来记录cab测试文件的结果。

如果你预期某个特定的错误会发生（比如说，你正在测试一段出错处理代码而不是测试是否对其进行了检测），可以使用--error num命令。这个命令向这个测试工具表明你已经预计到该错误会发生，即使它真的发生了，这次测试也要继续执行下去而不是半途中止。在编写测试代码的时候，这个命令应该放在可能会产生错误的那条命令的前面。你可以用逗号作为分隔符来给出多个出错代码。比如说，--error 1550, 1530表示这两个（假想的）错误是允许发生的，测试工作不会因为它们的发生而半途中止。

还可以在测试程序里使用控制代码流。比如说，可以使用一个循环来重复执行一组命令。如下所示的代码将重复执行一条命令100次：

```
let $1=100;
while ($1)
{
    # Insert your commands here
    dec($1)
}
```

另一个很有用的命令是sleep。sleep命令将按照其参数所给定的秒数暂停一会儿再执行下一条命令。比如说，--sleep 3.5告诉这个测试工具暂停3.5秒再执行下一条命令。如果你的网络速度比较慢或是测试因为服务器太忙而失败，这个命令会对你有帮助。sleep命令可以让测试慢下来，避免因为服务器或其他方面的性能低下而影响测试工作。

如果你想看到关于某个命令的更多信息，可以使用--enable_metadata命令。这个命令将生成并显示有关的内部元数据，它们在你为某个复杂的测试调试命令时会很有用。类似地，如果你想压缩输出的内容，可以用--disable_result_log命令暂时关闭记录功能，稍后再用--enable_result_log命令重新启用记录功能。

如果你的测试代码里有些命令会产生经过多次运行发生变化的数据（比如日期和时间字段），你可以通过--replace_column column string命令把它们替换为另一个字符串的办法来使这个测试工具忽略那些变化的值。比如说，如果你的第2列输出（数据列从1开始计数，不是0）是当前时间，你可以用命令--replace_column 2 CURTIME来使这个测试工具把下一条命令的第2列输出替换为字符串CURTIME。这虽然会让你看不到实际输出的值，但可以避免那些多次运行测试后发生改变而无法预料的价值影响你的测试。

最后，如果你需要在某个测试的内部引入额外的测试命令，可以用--source include/filetoinclude.inc命令从mysql-test/include子目录引入一个文件。在一些测试中，把一组比较常用的命令来构成一

个测试集，这在大型测试里很常见。

5. 报告bug

在运行某个测试或是创建你自己的测试时，你有可能会发现bug。MySQL AB公司欢迎大家对MySQL Test Suite测试工具的反馈并提供了一个报告bug的办法。不过，在发出这种电子邮件之前，你们应该先确认那确实是一个bug才行。

按照MySQL AB公司的要求，你应该把那个导致错误的测试单独拿出来运行并准确地找到那条引起问题的命令和出错代码。首先，重新安装MySQL或找个肯定没有问题的服务器来运行测试，检查这些错误是不是因为你的软硬件环境引起的（《MySQL参考手册》中的2.12节列出了一些这方面可能出现的问题，详见<http://dev.mysql.com/doc/refman/5.1/en/operating-system-specific-notes.html>）。你还应该以手动方式运行那个测试里的命令以确认那个错误和它的出错代码。有时候，以手动方式运行命令可以让你发现一些用其他办法无法获得的信息。在调试模式下运行服务器也很有帮助。最后，如果那个测试和出错条件可以重复，你应该在提交bug报告时把有关的测试文件、测试结果、测试拒绝文件、测试时使用的数据都收录进来^①。

4.2.2 MySQL 基准测试

MySQL Benchmarking Suite是MySQL AB公司面向大众推出的一套功能强大的基准测试工具。这套基准测试软件由一系列用来测试系统性能的Perl模块和脚本构成。MySQL的二进制和源代码发行版本几乎都收录有这个工具，它还可以在Windows平台上运行^②。在安装好MySQL之后，可以在其安装目录下的sql-bench子目录里找到一个名为run-all-tests.pl的Perl脚本。这个工具非常适合用来进行回归测试，随时都可以运行有关的测试程序去测试系统在当前条件下的性能。从MySQL的开发支持网站（<http://dev.mysql.com>）也可以下载到这套基准测试软件适用于各主流操作系统的版本。

与绝大多数基准测试工具一样，MySQL Benchmarking Suite最适合用来评估系统和环境变化所带来的影响。与MySQL Test Suite只能用来测试MySQL系统不同，MySQL Benchmarking Suite还可以用来测试其他品牌的数据库系统的性能。你完全可以用这个基准测试工具在你的MySQL、Oracle和Microsoft SQL Server系统上运行同样的测试，这么做可以让你准确地知道MySQL比你目前使用的数据库系统好多少。如果你想在其他品牌的数据库系统使用这个基准测试工具，可以使用--server='server'开关，这个参数的可取值包括MySQL、Oracle、Informai和MS-SQL。

这个工具提供了许多命令行参数供你选用以控制其执行情况，表4-2列出了几个最常用的参数并对它们进行了说明。关于这些命令行参数的更多信息可以在sql-bench子目录中的README文件里找到。

要想使用这个基准测试工具，只要进入MySQL安装目录下的sql-bench子目录并执行perl run-all-tests命令即可。你将注意到这个基准测试工具的一个重要特点：各项测试是按顺序运行的，每次只运行一个测试。如果你想测试多进程或多线程的性能，则需要使用第三方基准测试工具，如Super Smack或mybench。

① 你一定能赢得那台iPod！

② 需要安装ActivePerl，它是Windows平台上的官方Perl发行版本。详细情况以及下载最新的版本请参见www.active-state.org。

表4-2 性能测试工具的命令行参数

命令行参数	说 明
--log	把性能测试的结果保存到指定的文件里。用--dir选项来指定用来存放才能测试结果的子目录。结果文件是用Unix命令uname -a的输出来命名的
--user	用来登录服务器的用户名
--password	用来登录服务器的口令字
--host	服务器的主机名
--small-test	“小测试”模式，只运行最少的性能测试。省略这个参数将执行这个性能测试工具里的全部测试。对大部分用户来说，“小测试”模式已足以确定比较常用的性能指标

这个基准测试工具的另一个局限性是它目前还不具备可扩展性。换句话说，你无法为你自己的应用创建基准测试。不过，创建自己的基准测试对那些精通Perl语言的程序员来说并不是什么难事，因为它的源代码是免费公开的。如果你真的创建了自己的测试，别忘了把它拿出来与全世界的程序员共享。要知道，说不定有人正需要你创建的测试呢。

Super Smack和mybench

Super Smack是一个为MySQL开发的基准测试、强度测试和负载生成工具，与Apache Bench Tool很相似。它目前只能在少数几种Unix和Linux平台上运行。Super Smack可以从<http://vegan.net/tony/supersmack/>找到。mybench相对要简单一些，它是为MySQL开发的一个可定制的基准测试框架。它是用Perl语言编写的，可以从<http://jeremy.zawodny.com/mysql/mybench/>找到它。

提示 为了获得最好的效果，在进行基准测试之前应该先把MySQL的查询缓存功能禁用掉。在MySQL的客户接口里发出SET GLOBALS query_cache_size = 0;命令就可以关闭它的查询缓存功能。这将使你的基准测试把查询命令的实际执行时间，而不是系统从查询缓存里快速检索出与之相匹配的查询结果所花费的时间记录下来。

如果这个性能工具的“小测试”模式已足以满足你的需要，可以运行perl run-all-tests-small-test命令来生成一份基本的性能测试结果报告。运行所有的测试可以对系统的性能做出更全面的评估，但那些测试需要花费相当长的时间才能全部完成。如果你只想了解系统的某个特定部分的性能，你可以把有关的测试单独挑出来执行。比如说，如果你只想测试服务器的连接性，只需执行perl test-connect就可以了。表4-3列出了一些比较常用的单项测试。

表4-3 基准测试的部分列表

测 试	说 明
test-ATIS.sh	创建29个数据表并对它们进行一些查询
test-connect.sh	测试服务器的连接速度
test-create.sh	测试数据表的创建速度
test-insert.sh	测试数据表的创建和填充操作
test-wisconsin.sh	运行这个性能测试工具的PostgreSQL版本

注解 性能测试工具是在一个单线程里运行那些测试的。MySQL AB公司已经计划在这个性能测试工具的未来版本里增加一些多线程测试。

为MySQL开发的基准测试工具还有很多，如果你想了解更多关于这方面的信息，请参考Michael Kruckenberg和Jay Pipes合著的*Pro MySQL*^①一书，那本书对MySQL的各个方面都做了详细的分析和介绍。

1. 运行小测试

现在，我们一起去看看在你的系统上运行这个基准测试工具会得到些什么。在下面的例子里，我将在Windows系统上以“小测试”模式运行这个基准测试工具。代码清单4-5是在测试时生成的输出文件的开头部分。

代码清单4-5 “小测试”模式下的性能测试（节选）

```
D:\source\C++\mysql-5.1.9-beta\sql-bench>perl run-all-tests --small-test
```

```
Benchmark DBD suite: 2.15
Date of test:      2006-05-21 23:12:16
Running tests on:  Windows NT 5.1 x86
Arguments:        --small-test
Comments:
Limits from:
Server version:   MySQL 5.1.9 beta/
Optimization:     None
Hardware:

alter-table: Total time: 4 wallclock secs ( 0.05 usr  0.01 sys +
0.00 cusr  0.00 csys = 0.06 CPU)
ATIS: Total time: 6 wallclock secs ( 1.33 usr  0.28 sys +
0.00 cusr  0.00 csys = 1.61 CPU)
big-tables: Total time: 0 wallclock secs ( 0.14 usr  0.01 sys +
0.00 cusr  0.00 csys = 0.15 CPU)
connect: Total time: 4 wallclock secs ( 0.69 usr  0.39 sys +
0.00 cusr  0.00 csys = 1.08 CPU)
create: Total time: 1 wallclock secs ( 0.02 usr  0.00 sys +
0.00 cusr  0.00 csys = 0.02 CPU)
insert: Total time: 11 wallclock secs ( 2.59 usr  0.67 sys +
0.00 cusr  0.00 csys = 3.27 CPU)
select: Total time: 16 wallclock secs ( 4.06 usr  0.45 sys +
0.00 cusr  0.00 csys = 4.52 CPU)
transactions: Test skipped because the database doesn't support transactions
wisconsin: Total time: 15 wallclock secs ( 2.66 usr  0.44 sys + 0.00 cusr  0.00
csys = 3.10 CPU)
```

All 9 test executed successfully

① Apress公司2005年出版。

在这份代码清单的开头，可以看到一些关于本次基准测试的元数据，其中包括测试日期、操作系统的版本、服务器的版本以及任何特殊的优化措施或专用硬件（虽然我的Windows系统上没有）。在这些元数据的后面，会看到按实际花费时间（单位是秒）统计出来的各个单项测试结果。请注意，wallclock的字面意思是“墙上的挂钟”，它指的是实际经过的时间，但这里面还包括了网络传输延迟之类的东西。括号里的数字是这个性能测试工具本身在执行时花费的时间，用wallclock时间减去它们才是各项测试所花费（包括传输和处理）的时间。你用不着过分关心这些统计结果，它们只是为了让人们能够迅速了解本次测试的基本情况才放在那儿的。接下来的部分才是最值得关心的地方，即各项测试的实际执行用时。表4-4是我在Windows系统上进行基准测试得到的结果，为了节约篇幅，我对它做了一些删节。

表4-4 “小测试”模式下各单项测试的结果（每个操作的总记录）

操 作	总时间	usr	sys	cpu	测试的次数
alter_table_add	3.00	0.01	0.00	0.01	92
alter_table_drop	1.00	0.02	0.01	0.03	46
connect	0.00	0.08	0.11	0.19	100
connect+select_1_row	1.00	0.09	0.09	0.19	100
connect+select_simple	1.00	0.08	0.03	0.11	100
count	1.00	0.02	0.00	0.02	100
count_distinct	1.00	0.05	0.00	0.05	100
count_distinct_2	0.00	0.00	0.00	0.00	100
select_range	1.00	0.08	0.03	0.11	41
select_range_key2	1.00	0.11	0.00	0.11	505
select_range_prefix	0.00	0.11	0.02	0.12	505
select_simple	0.00	0.05	0.00	0.05	1000
select_simple_cache	0.00	0.06	0.03	0.09	1000
select_simple_join	0.00	0.05	0.00	0.05	50
update_big	0.00	0.00	0.00	0.00	10
update_of_key	0.00	0.02	0.02	0.03	500
update_of_key_big	0.00	0.00	0.00	0.00	13
update_of_primary_key_many_keys	0.00	0.00	0.00	0.00	256
update_with_key	1.00	0.16	0.02	0.17	3000
update_with_key_prefix	1.00	0.09	0.02	0.11	1000
wisc_benchmark	2.00	0.97	0.14	1.11	34
总计	56.00	11.45	2.19	13.58	78237

在进行基准测试的时候，我喜欢把测试报告的后半部分转换为一个电子表格以便对测试结果进行统计分析。这还可以让我方便地对预期结果、“事前”结果和“事后”结果进行对比和计算。表4-4给出了各项操作所花费的总时间、这个基准测试工具本身所花费的时间（usr、sys、cpu）和每项操作的测试次数。

表4-4的最后一行对各列数据进行了统计，这可以让我们知道基准测试所花费的总时间。表4-4和代码清单4-1共同构成了我的Windows系统的性能基准线。我建议你针对自己的数据库服务器进行基准测试时，也像我这样把重要的信息保存为一份历史档案。

2. 进行单项测试

假设你想测试一下数据表创建操作的性能。从表4-3里可以查知，这个测试的名字是test-create。为了进行这个测试，你需要进入sql-bench子目录并输入perl test-create命令。代码清单4-6是在我的Windows系统上运行这条命令的结果。

代码清单4-6 单项性能测试：test-create测试的输出

```
D:\source\C++\mysql-5.1.9-beta\sql-bench>perl test-create

Testing server 'MySQL 5.1.9 beta/' at 2006-05-22 21:47:51

Testing the speed of creating and dropping tables
Testing with 10000 tables and 10000 loop count

Testing create of tables
Time for create_MANY_tables (10000): 154 wallclock secs ( 2.22 usr
0.34 sys + 0.00 cusr 0.00 csys = 2.56 CPU)

Accessing tables
Time to select_group_when_MANY_tables (10000): 41 wallclock secs ( 0.91 usr
0.16 sys + 0.00 cusr 0.00 csys = 1.06 CPU)

Testing drop
Time for drop_table_when_MANY_tables (10000): 46 wallclock secs ( 1.19 usr
0.25 sys + 0.00 cusr 0.00 csys = 1.44 CPU)

Testing create+drop
Time for create+drop (10000): 130 wallclock secs ( 3.28 usr 0.47 sys +
0.00 cusr 0.00 csys = 3.75 CPU)
Time for create_key+drop (10000): 132 wallclock secs ( 3.08 usr 0.66 sys +
0.00 cusr 0.00 csys = 3.73 CPU)
Total time: 503 wallclock secs (10.69 usr 1.88 sys +
0.00 cusr 0.00 csys = 12.56 CPU)

D:\source\C++\mysql-5.1.9-beta\sql-bench>
```

在代码清单4-6里，可以看到各项操作的典型参数。请注意，每项操作都反复执行了许多次。这是为了保证测试结果不会受到偶发事件的影响所必需的，这可以让测试结果更能真实地反映出系统的实际情况。

我选择这个例子的目的是想让大家了解一下性能测试的另一种用途。假设你想对SQL语言里的CREATE命令进行修改或是创建一个全新的CREATE命令。此时，可以通过修改test-create脚本的办法把对新CREATE命令的测试也包括进来。然后，运行有关的基准测试并为你新命令建立一个基准线。在对MySQL系统进行扩展的时候，这个办法可以让你少走许多弯路。如果你很关心你系统的性能或扩展性、或者如果你真的打算扩展MySQL系统，建议你好好试试这个办法。

3. 应用基准测试

在继续进行解释之前，我想再次讨论一下这个主题，因为这对了解和体验基准测试的好处很重要。

要想让基准测试发挥作用，唯一的办法是把各次基准测试的结果保存为一份历史档案。我是这么做的：把每次基准测试的结果集中保存在一个以测试日期命名的子目录里，我会在这个子目录下为各单项基准测试分别创建一个子目录来存放其输出文件（通过--log参数）。除了输出文件，我还会把系统和环境的当前配置情况（你可以用一个系统调查软件来生成一份这样的文件）以及一份简短的说明也记录并保存起来。

此后，如果我把系统的性能与过去某个时候的情况进行对比。比如说，当我修改了一个服务器变量并想看到它对系统性能的影响时，我会在做出修改之前和之后分别进行一次基准测试，然后从我的系统基准测试历史档案里找出系统状态最稳定时期的测试结果与这些结果进行对比。这个办法还可以让我了解系统性能随时间而变化的情况。

像这样来使用基准测试会让你的系统管理水平达到一个很少有人达到的高度。

4.2.3 MySQL 性能分析

MySQL的服务器工具包（或源代码发行版本）没有提供一个正式的性能分析工具，但我们可以利用它提供的许多诊断工具来进行简单的性能分析。比如说，你可以检查线程的执行状态，查看服务器的日志，甚至可以观察到优化器是如何执行一条查询命令的。

如果你想查看当前线程的清单，可以使用MySQL提供的SHOW FULL PROCESSLIST命令。这个命令将把所有正在运行的进程或线程、运行它们的用户、用户所使用的客户机、它们用到的数据库、当前命令、执行时间、状态参数以及线程提供的其他信息显示在屏幕上。比如说，如果我在系统上运行这个命令，将看到如代码清单4-7所示的结果。

代码清单4-7 SHOW FULL PROCESSLIST命令的输出

```
mysql> SHOW FULL PROCESSLIST \G
```

```
***** 1. row *****
  Id: 7
  User: root
  Host: localhost:1175
  db: test
  Command: Query
  Time: 0
  State: NULL
  Info: SHOW FULL PROCESSLIST
1 row in set (0.00 sec)
```

从上面这份清单可以看出，系统现在只有我一个用户（root），是从本地主机登录的，执行了一条查询命令，执行时间是0，还执行了我当前正在执行的命令。这个命令的不足之处是它的输出只是某一时刻的快照，你必须反复运行多次才有可能找出性能瓶颈的发生规律。还好，有个名为mytop工具可以帮上忙，它可以反复调用这个命令并把数据显示成几种更容易查看的视图。关于mytop工具的更多信息和mytop下载可以在Jeremy Zawodny的网站（<http://jeremy.zawodny.com/mysql/mytop>）上找到。

注解 mytop工具在Windows平台上已取得了有限的成功。

另一条可以用来查看服务器信息的命令是SHOW STATUS命令。这个命令将显示所有的服务器变量和状态变量。可以想象，那会是一份非常长的清单。还好，你可以给这条命令加上一个LIKE子句来限制这份清单的长度。比如说，你只想查看与线程有关的信息，输入SHOW STATUS LIKE "thread%";即可。代码清单4-8是这个命令的输出示例。

代码清单4-8 SHOW STATUS命令

```
mysql> SHOW STATUS LIKE "threads%";
```

Variable_name	Value
Threads_cached	0
Threads_connected	1
Threads_created	6
Threads_running	1

4 rows in set (0.00 sec)

要想查看慢查询日志（slow query log），你可以设置log-slow-queries变量并用long_query_time变量设置查询等待时间。慢查询的等待时间没有统一的标准，你需要根据你自己的经验去判断需要等待多长时间的查询才算是慢查询。如果你想查看慢查询，可以使用mysqldumpslow命令去显示慢查询。这个命令将把慢查询分门别类地列出来，这条命令还可以用来查看其他一些信息，其中包括锁定信息、预期检索出多少个数据行、实际检索出多少个数据行、各有关操作步骤花费了多少时间，等等。

通用查询日志（general query log）可以用MySQL Administrator软件来查看。如果你是从本地连接到服务器的，可以查看到所有的日志。如果你以前从没用过MySQL Administrator软件，建议你现在就从<http://dev.mysql.com/downloads>网站把它下载下来试试。

提示 可以用MySQL Administrator软件对MySQL服务器的几乎每一个方面进行调控，其中包括启动设置、日志功能和变量。

MySQL系统提供的最后一个性能分析功能是可以让你检查它的优化器是如何执行查询命令的能力。虽说这个功能不是为了测评性能而开发的，但我们可以用它来诊断慢查询日志里的查询为什么执行得那么慢。通过一个简单的例子，我们来看看优化器将如何执行下面这个查询：

```
select * from customer where phone like "%575%"
```

这个查询使用了LIKE子句，还使用了百分号（%）来给出数据值，这总会导致服务器在检索数据时不使用索引。如果你在执行这个命令时在它的前面加上一个EXPLAIN关键字，你将看到优化器提供的改进建议。代码清单4-9给出了使用EXPLAIN命令的结果。

代码清单4-9 EXPLAIN命令的输出

```
mysql> explain select * from customer where phone like "%575%" \G
```

```
***** 1. ROW *****
```

```
id: 1
select_type: SIMPLE
table: customer
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 599
Extra: Using where
1 row in set (0.00 sec)
```

这份输出表明：该命令是customer表上的一个简单的选择操作，没有可供使用的键，表里有599条记录，优化器使用的是where子句。具体到这个例子，这次执行是一次不使用索引的表全表扫描操作——而这种SELECT语句可以说是速度最慢的。

4.3 小结

本章介绍了一些软件测试技术和测试策略，讲述了软件测试的好处以及如何把测试驱动开发融合到你的软件项目当中。本章还对测试MySQL的几种测试工具进行了介绍。重点展示了MySQL测试工具和基准套件，还介绍了几个MySQL性能分析脚本。

具备这些测试工具的知识可以确保你对MySQL源代码尽可能做出高质量的修改。只有掌握了这些知识，你对MySQL系统的扩展或增强才会达到MySQL AB公司所坚持的高质量标准^①。既然你们已经知道了这一点，现在就可以开始设计你的解决方案并在设计阶段就把测试工作安排妥当了。

此后为本书第二部分，下一章将介绍软件开发工具当中最为重要一种：调试！

^① 如果不是为了这个目的，它就不会推出这么多的测试、基准测试和性能分析工具了！

Part 2

第二部分

扩展 MySQL

本书的第二部分将采用实践的方式把剖析和扩展MySQL系统所需要的工具介绍给大家，介绍如何修改MySQL源代码和如何把MySQL用作一个嵌入式数据库系统。第5章介绍的各种调试技巧和技术有助于保证开发工作的顺利进行，减少不必要的错误和麻烦。在介绍各种调试技术时，我们还对它的优点和缺点进行了分析。第6章指导你如何把MySQL系统嵌入企业级应用程序。第7章对MySQL的插件式存储引擎机制进行了剖析，可以参考这一章的示例和项目来创建自己的存储引擎。第8章给出了一些最为流行的MySQL代码修改示例。你将学到如何修改SQL命令来添加新的参数和函数，以及如何添加新的SQL命令。

本部分内容

- 第5章 调试
- 第6章 嵌入式 MySQL
- 第7章 创建自己的存储引擎
- 第8章 为 MySQL 添加函数和命令

本章将讨论每一个开发人员都必须掌握的强大工具之一：调试。良好的调试技能有助于保证软件项目开发更容易，减少不必要的错误和麻烦。文中还将介绍几种最常用的MySQL调试技术。如果你的调试技能已经足够扎实，可以跳过下面两节而直接进入5.3节。

5.1 调试介绍

只要你编写过比“Hello world”更复杂的程序，就肯定知道什么是缺陷（bug）。绝大多数缺陷很容易被发现，但有些查找和纠正起来相当困难。

如果让你把“调试”这个概念解释给一名程序员新手，你大概会告诉他说这是一个排除故障的过程，目的是发现什么问题，你可能还会提示他良好的调试技能来自对调试技术和调试工具的熟练掌握。这是一个很不错的开场白，但如果你想让那位新手对软件调试工作的细节有更深入的理解，就得花上不少的时间了。

对于初学者来说，在动手之前先花些时间把想要查找和纠正的缺陷大致是什么样的弄清楚是很重要的。缺陷基本上可以分为两种：语法错误和逻辑错误。语法错误在代码编译时会被编译器发现并报告出来，虽然有些语法错误很难纠正，但你不纠正它们就无法通过编译，就得不到可执行的软件。逻辑错误是那些在编译时无法发现，等到软件运行时才反映出来的错误。调试就是在程序里找出并纠正错误的过程。

注解 有许多工具可以在编译阶段（或更早）用来查找代码中的错误，这些工具可以降低逻辑错误的风险。它们有的比较简单，比如一个简单的用来查找“无效”代码的流控制解析器；有的相当复杂，比如一个复杂的取值范围和类型检查器，它可以检查代码里是否存在数据不匹配的情况；还有一些工具通过运用代码加固的最佳实践方法来检查代码中的出错处理部分。

内部存在逻辑错误的系统往往会表现出这样或那样的奇怪行为或产生错误的数据。在极端情况下，整个系统都有可能崩溃。一般来说，结构设计良好、又根据大家公认的代码加固规则进行过代码优化的系统比其他系统更健壮，它们往往可以在错误发生时及时捕获错误并做出妥善处理，尽管如此，系统也有可能因为某个错误过于严重而崩溃（或是被操作系统终止）。从某种意义上讲，崩溃并不是最坏的情况，因为它还可以保护数据和系统状态，而继续带病运行造成的损失会更大。

软件调试技术的水平高低体现在你能不能快速地——通过观察系统状态的变化情况也好、直接分

析有关代码和数据也罢——找到问题的根源。人们把用来进行调试的软件工具称为系统调试器(System debugger)。接下来的几节将介绍几种常见的调试技术和相关的调试器。

调试的起源

你肯定知道关于计算机“虫子(bug)”的多个版本的故事,我想讲一个我最喜欢的故事。我曾有幸在距离海军上将Grace Hopper发现第一个计算机“虫子”的不远的地方工作,据说,当时(1945年)还是海军少将的Hopper正在使用的一台名为Mark II Aiken Relay Calculator的大型计算机出了问题,在经过一番查找之后,Hopper在一个烧毁的继电器里发现了一只死蛾子,在更换那个继电器之后,系统恢复了正常。于是,Hopper在报告里写到:故障的原因是一只虫子,解决办法是捉住了虫子。从那时起,消除代码隐患的工作就被人们称为“捉虫子(debugging)”。

5.2 调试技术

每个程序员都有自己的一套调试办法,就好像每个人捉虫子的方法都不太一样。不过,这些方法大致可以分为几大类型。

最基本的方法是在源代码里加入一些语句,这些语句将成为最终的可执行文件的一部分。这包括嵌在代码里的调试语句(在执行时把有关的消息和变量值打印出来,如printf("Code is at line 199. my_var = %d\n", my_var);)和出错处理器(error handler)。绝大多数开发人员会把这些技巧当作最后一招(当缺陷不容易找到时)或是在开发阶段使用(一边编写代码,一边进行测试)。你或许会认为出错处理器与健壮性和加固的关系要比与调试的关系更大,其实它们也可以是强大的调试工具。因为这类方法把调试代码嵌在了程序里,你可以在调试工作结束后,用条件编译指令把它们从最终的可执行文件里去掉,但也有许多程序员会把调试语句留在代码里让它们成为程序的一部分。在使用这个技巧的时候,一定要保证你的调试代码不会给程序带来负面影响。

人们最熟悉的调试技术应该是使用各种外部调试器了。外部调试器(external debugger)的“外部”是指它们与被调试的程序相互独立,彼此没有内在的联系。它们既可以用来实时监控系统的运行情况,也可以让你控制代码的执行情况并在任何地方中止和开始代码的执行。在后面的几节里将对这类技术做更详细的介绍。但首先来看看调试工作的基本过程。

5.2.1 基本过程

每次调试都是一次与众不同的经历,但其过程却总是遵循那几个基本的步骤。坚持按照这些基本步骤来进行调试可以让调试工作更有效率且更有回报。没有什么比在奋战几个小时之后找出bug的感觉更好了。你可能早已在工作中形成了自己的套路,但我想它们至少应该包括以下几个步骤。

- (1) 确定缺陷确实存在(bug报告、测试)。
- (2) 重现缺陷。
- (3) 编写测试来确认那个缺陷。
- (4) 分离出缺陷的原因。
- (5) 制作相应的补丁并应用它。

(6) 运行测试来验证缺陷已被修复：是，继续；否，返回第4步。

(7) 运行回归测试，确认你的补丁不会对系统的其他部分造成影响。

有时候，确定缺陷确实存在并不是件容易的事情。在面对缺陷报告时，不管它是一份官方的bug报告还是一次未能通过的系统测试，千万不要等闲视之，尤其是那些不起眼的缺陷。会导致系统崩溃或数据丢失的大问题当然会引起你的注意，那些偶然发生或只在特定条件下才会发生的缺陷还会这样吗？在遇到这种情况的时候，应该首先假设缺陷确实存在。

如果你运气好，收到一份完整的bug报告，描述了如何重现缺陷，你可以据此编写一个测试来确认缺陷的存在。如果收到的bug报告语焉不详，就可能要花上许多时间才能把问题本身搞清楚。

一旦你能重现缺陷，就应该立刻编写一个测试，看看是否能再次产生这个问题。这个测试很重要，在你修补了那个缺陷之后，还需要用它来证明你已经解决了那个问题。

下一步是调试工作的正式开始：分离出缺陷的原因。本章将要介绍的调试技术就是在这个步骤派上用场的，你必须想尽一切办法排除其他因素，诊断出问题的真正根源。这一步在软件调试工作中最重要，也最具有挑战性。

对程序员来说，为弥补某个缺陷而编写补丁（有时也叫修复）和编写程序一样，通常都是一个迭代过程。我只想提醒大家注意一点：每次只修改一个地方，然后立刻测试它对缺陷和系统其他部分的影响。等你认为已经彻底修补好那个缺陷之后，应该再次运行你在步骤(3)里编写的测试程序来证明确实解决了问题。如果这次测试“失败”了（没有找到缺陷），就表明你达到了解决问题的目的！如果这次测试“成功”了（故障现象依旧），你应该回到步骤(4)重新开始这个过程直到你的测试“失败”为止。

如何创建和使用补丁

补丁是一个人们不怎么熟悉的软件开发技术，其实它不过是包含初始文件（有缺陷）和修改后的文件（无缺陷）版本之间的差异的文件。创建补丁的时候，需要运行一个名为diff的GNU程序并把它的输出保存为一个文件。（可以从www.gnu.org/software/diffutils/diffutils.html上找到diff程序。它目前只有适用于Unix和Linux平台的版本，但Windows用户可以在Cygwin环境下使用它。）例如，如果你修改了mysqld.cc文件——给它增加了一行用来改变版本号的代码，你可以用diff-Naur `mysqld.cc.old mysqld.cc > my_sqld.patch`命令来为这段代码创建一个补丁。这条命名令将建出一个内容如下所示的文件：

```
--- mysqld.cc.old 2006-08-19 15:41:09.000000000 -0400
+++ mysqld.cc 2006-08-19 15:41:30.000000000 -0400
@@ -7906,6 +7906,11 @@
 #endif
     if (opt_log || opt_update_log || opt_slow_log || opt_bin_log)
         strmov(end, "-log"); // This may slow down system
+/* BEGIN DBXP MODIFICATION */
+/* Reason for Modification: */
+/* This section adds the DBXP version number to the MySQL version number. */
+ strmov(end, "-DBXP 1.0");
+/* END DBXP MODIFICATION */
 }
```


你还可以用diff命令为一组文件或是一个完整的子目录创建一个差异文件。有了这个差异文件，你就可以拿到其他机器上去打补丁了。

在打补丁的时候，需要用到一个名为patch的GNU程序。（可以在www.gnu.org/software/patch网页上找到patch程序。它目前只有适用于Unix和Linux平台的版本，但Windows用户可以在Cygwin环境下使用它。）patch程序将从diff程序生成的补丁文件里读取信息并给有关的文件打上补丁。比如说，如果你想给mysqld.cc文件打上刚才用diff程序创建的补丁，只须执行patch < mysqld.patch命令即可；patch程序将根据mysqld.patch文件里的信息对你的mysqld.cc文件做出必要的修改。

利用补丁来给文件做一些小修小补是很方便的——修补漏洞就是一个典型的例子。在修补了一个漏洞之后，你可以创建一个补丁并把它拿到其他机器上对有关的文件做出同样的修改。

有许多开源项目都利用了“打补丁”的概念来交流各种改动。事实上，补丁正是世界各地的程序员为MySQL源代码添砖加瓦的主要手段——你用不着上传整个文件，只要把相关的补丁发送给MySQL AB公司就行了。MySQL AB公司会在收到补丁之后对它进行必要的检查并做出“接受”（并打上补丁）或“拒绝”的决定。如果你还从未用过diff和patch程序，我建议你赶快把它们下载下来试试。

5

最后，在修补好缺陷之后，还应该进行一次回归测试以保证你的补丁没有给系统带来其他的问题。如果你修补的系统采用的是组件或模块化的体系结构，并有着完备的文档，你可以根据它的需求矩阵轻而易举地把相关的组件或模块找出来。需求矩阵（requirements matrix）可以跟踪来自用例、类和序列图的需求，还能识别为这些需求所创建的测试。因此，在修改了某个类或模块的一部分之后，可以很容易地找到一组测试来进行回归测试。如果你手里没有需求矩阵，可以用一个简单的文档或电子表格来创建一个，或是用源代码所满足的需求来为它们添加注释。

5.2.2 内嵌调试语句

有相当一部分程序员喜欢在他们编写的代码里加上一些打印语句来帮助调试，这些语句的打印内容可以是某个变量的值、也可以是一条消息。有些人认为使用内嵌调试语句的调试技术不够成熟或不够简便，这种认识是不全面的。内嵌调试语句是比较麻烦，但它们的作用却不可低估。广义上讲，能够在程序运行时输出一些信息、数据或系统状态的代码都可以称为内嵌调试语句。

在给出一个内嵌调试语句的例子之前，我想先和大家说说使用内嵌调试语句会给系统带来哪些影响。首先，这些调试语句也是代码！如果它们除了把一些信息写到标准错误流（窗口）外还做了其他的事情，就会给系统带来一系列影响。其次，内嵌调试语句往往会在编译阶段被程序员利用条件编译指令去掉，有些人认为这同样会给系统带来一些难以预料的影响，因为你所编译的系统不是你曾经调试过的系统！

有时候，客观条件不允许你使用外部调试器，或是有些故障只是偶尔才会发生^①。在遇到这类情况时，内嵌调试语句就派上用场了。可能发生这种事情的例子包括：实时系统，多进程和多线程系统，以及处理大量数据的大型系统等。

^① 我个人并不认为是偶尔发生。计算机只是执行人的命令的机器，除非它们可以自己思考。

插 桩

许多人认为内嵌调试语句是一种插桩 (instrumentation) 形式。这个概念涵盖了用来追踪系统性能、数据、用户、客户端和执行情况的各种代码。在编程时, 通常会在代码中加入若干语句来显示数据值、警告消息和出错消息等, 从而实现插桩。但在一个沙箱环境里用来监控系统运行情况的大段代码 (它们通常是一些比较大的封装函数或打包器) 甚至整个程序也属于这个范畴, 英特尔的 Pin 就是这样一个软件插桩工具。如果你想了解更多关于软件插桩和 Pin 的信息, 请访问 <http://rogue.colorado.edu/Pin/docs/tutorials/AsplosTutorial.htm>。

内嵌调试语句有两种。第一种侧重于收集信息 (如内存状态和变量的值等)。这类调试语句多用于开发阶段, 在编译阶段会被程序员改成注释或是用条件编译指令去除。第二种侧重于跟踪系统的执行路径, 这类调试语句可以用在任意场合, 程序员通常会设置一个开关, 以便在程序运行时启用或禁用它们。因为第一种内嵌调试语句对绝大多数程序员来说都很熟悉 (很多人就是这样学习调试技术的), 所以接下来我将通过一个例子来直接讨论第二种内嵌调试语句。

假设你有一个运行在多线程模型下的大系统, 现在需要找出某个缺陷的根源。用一些内嵌调试语句去查看内存或变量值可能会有所帮助, 但编程漏洞往往不会这么容易被发现。此时, 你需要了解这个缺陷是在系统处于何种状态下产生的。如果你在该系统的源代码里安插了一些调试语句, 让它们把进入和离开某个函数的情况 (还可以有一些关于数据的额外信息) 记录到一个日志文件中, 你就可以通过查看这个日志文件而了解故障发生时系统所处的状态了。代码清单5-1是一段包含着内嵌调试语句的MySQL源代码, 我已经用黑体字把那些调试语句凸显出来了。这段代码里的内嵌调试语句会把一些信息写入一个踪迹文件, 你可以在系统执行结束 (或崩溃时) 查看该文件。

代码清单5-1 内嵌调试语句示例

```

/*****
** List all Authors.
** If you can update it, you get to be in it :)
*****/

bool mysql_d_show_authors(THD *thd)
{
    List<Item> field_list;
    Protocol *protocol= thd->protocol;
    DBUG_ENTER("mysql_d_show_authors");

    field_list.push_back(new Item_empty_string("Name",40));
    field_list.push_back(new Item_empty_string("Location",40));
    field_list.push_back(new Item_empty_string("Comment",80));

    if (protocol->send_fields(&field_list,
                             Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DBUG_RETURN(TRUE);

    show_table_authors_st *authors;
    for (authors= show_table_authors; authors->name; authors++)

```

```

{
    protocol->prepare_for_resend();
    protocol->store(authors->name, system_charset_info);
    protocol->store(authors->location, system_charset_info);
    protocol->store(authors->comment, system_charset_info);
    if (protocol->write())
        DEBUG_RETURN(TRUE);
}
send_eof(thd);
DEBUG_RETURN(FALSE);
}

```

请注意，代码清单5-1里的第一条内嵌调试语句会在系统执行到这个函数时，把这个函数的名字写入踪迹文件，在这个函数的每一个退出点都有一条类似的调试语句把这个函数的名称和返回值记录下来。代码清单5-2节选自执行SHOW AUTHORS命令时得到的踪迹文件。为了让大家看到SHOW AUTHORS命令执行成功时会留下什么样的踪迹，我删去了这个清单的大部分内容。

代码清单5-2 踪迹文件示例

```

T@6 : | | | >mysqld_show_authors

...

T@6 : | | | | >send_eof
T@6 : | | | | packet_header: Memory: 0x9b6ead8 Bytes: (4)
05 00 00 50
T@6 : | | | | | >net_flush
T@6 : | | | | | >vio_is_blocking
T@6 : | | | | | | exit: 1
T@6 : | | | | | <vio_is_blocking
T@6 : | | | | | >net_real_write
T@6 : | | | | | | >vio_write
T@6 : | | | | | | | enter: sd: 17776, buf: 0x0734D278, size: 5029
T@6 : | | | | | | | exit: 5029
T@6 : | | | | | | <vio_write
T@6 : | | | | | <net_real_write
T@6 : | | | | | <net_flush
T@6 : | | | | | info: EOF sent, so no more error sending allowed
T@6 : | | | | <send_eof
T@6 : | | | <mysqld_show_authors

```

注解 在默认的情况下，这些内嵌调试语句是不工作的。要想启用它们，你需要在编译这个服务器时用上debug选项，还要用--debug命令行选项来让服务器运行在调试模式下。这样才能启用那些内嵌调试语句并为它们创建一个踪迹文件。在Linux系统上，这个踪迹文件是/tmp/mysqld.trace；在Windows系统上，这个文件是c:\mysqld.trace。请注意，因为MySQL里的所有函数都或多或少地有几条内嵌调试语句，所以这些文件可能会变得非常大。

这个技巧虽然简单，用处却很多。在查看踪迹文件里记录的系统执行流程时，你往往会发现一些

值得进一步追查的东西，而有时仅知道去哪找东西都是一个巨大的挑战。

5.2.3 出错处理器

在使用软件的时候有没有见过出错消息？不管你使用的是商业软件还是开源软件，绝大多数出错消息都来自某个出错处理器。

你们也许会对我把出错处理器当作一种调试技巧来讨论感到奇怪。原因很简单，一个好的出错处理器会给出问题的原因和可能的改正选项。好的出错处理器可以提供足够的信息供程序员了解什么地方出了问题以及如何解决问题，有时还能提供一些可以帮助他们诊断故障原因的信息。令人遗憾的是，好的出错处理器并不多见，像图5-1这样的对话框——不知所云的出错消息、让人摸不着头脑的解决方案选项——倒是经常遇到。

用户几乎每天都可以看到这样的出错消息，但它们对一般用户来说几乎没有任何价值。很明显，编写这段代码的程序员没有从用户的角度考虑这个问题。系统的开发人员一看就明白的东西在普通用户眼里也许连废话都不如。在编写出错消息的时候，应该从用户的角度来思考，不仅应该把什么地方出了问题解释清楚，更应该提供一个解决办法（如果有的话），或至少应该为用户提供一个向你们报告问题的途径。要是还能把一些可以帮助程序员诊断问题的信息记录下来就更好了，这可以是一个日志文件、一份系统状态清单，或者一份自动生成的报告。图5-2给出了一个对用户友好的出错消息对话框。

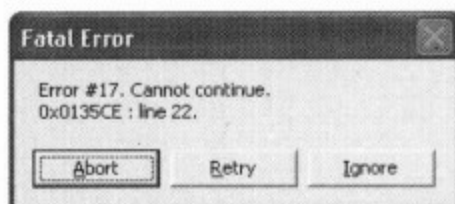


图5-1 糟糕的出错处理器示例

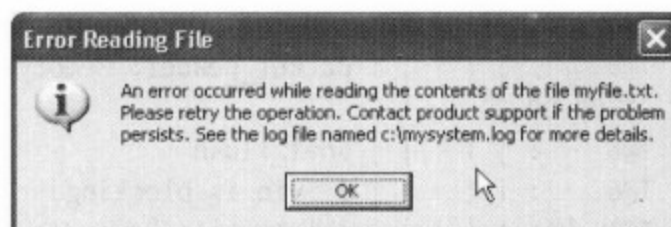


图5-2 良好的出错处理器示例

出错处理器并非只能显示出错消息。还有一种代码结构也叫作出错处理器，那是一些专门用来捕获和处理出错情况的代码块，C++语言里的try...catch语句就是这方面的典型例子。代码清单5-3给出了一个try ... catch语句块，它实际上是C++出错处理器（也叫作异常处理器）的基本语法。

代码清单5-3 C++出错处理器示例：try...catch语句

```
try
{
    //attempt file operation here
}
catch (CFileException* e)
{
    //handle the exception here
}
```

像C++这样有专用出错处理器语法的程序设计语言不算很多，但只要是支持条件语句的语言，都可以用来编写这样的出错处理器。比如说，代码清单5-4给出了一段用C语言写的示例代码，这段代码将对返回值进行检查，如果发生错误，则进行相应的处理。作为编写出错处理器的基本原则，在列

举出错条件时，一定要把可能发生的每一种情况都考虑周全；在处理出错情况时，至少应该保证不会影响系统的性能，不会导致数据丢失或损坏（这一点更重要）。

代码清单5-4 C出错处理器示例

```
if ((archive= gzopen(share->data_file_name, "rb")) == NULL)
{
    if (errno == EROFS || errno == EACCES)
        DEBUG_RETURN(my_errno= errno);
    DEBUG_RETURN(HA_ERR_CRASHED_ON_USAGE);
}
```

出错处理器并非只能显示出错消息，它们也是调试工作的一道防线。好的出错处理器不仅可以捕获和处理出错情况，还可以保存和显示诊断信息。

请再看看代码清单5-4。这段代码是从MySQL源代码的ha_archive.cc文件里节选出来的。请注意其中的黑体字部分，那是一条内嵌调试语句，像这样的语句在这个文件里还有很多。之所以选择这段代码作为例子，是因为这条内嵌调试语句出现在出错处理器里，可以让我们知道怎样才能把追查故障根源所需要的诊断信息记录下来。如果让我来调试这段代码，我会在调试模式下运行MySQL服务器并在踪迹文件里查看由这个出错处理器记录下来的诊断信息。

我建议大家参照这个例子来编写你们所有的出错处理代码。显示给用户看的出错消息必须让用户能看明白，与此同时，还应该把出错代码（返回值）捕获下来，并把它们以及相关的诊断信息记录在案。像这样编写和使用出错处理器，不仅可以提高调试技术水平，还可以让系统更容易诊断。有时候，甚至用不着运行调试器，踪迹文件里的诊断信息已经足以用来直接找出问题的根源了。

5.2.4 外部调试器

调试器是一种专门用来分析和跟踪程序代码执行情况的工具。绝大多数被称为调试器的工具与被调试的软件是相互独立的，只在调试时才会发生联系，因而严格地讲，应该称为外部调试器（external debugger）。不过，出于简明和习惯的考虑，在本小节的讨论中还是把它们统称为“调试器”好了。

调试器有很多种，但大致可以分为三个类型。人们最熟悉的调试器是一些独立存在并且可以独立运行的工具程序，你可以把它们与一个正在运行着的进程关联起来，并通过它们去调控那个进程的执行情况。还有一些调试器本身属于一个更大的软件工具的一部分，它们通常通过一个人机交互操作界面来接受调试命令并完成相应的控制和调查工作。最后是一些专用的调试器，它们可以对被调试系统进行一些更高级的调控。我将在接下来的几个小节里对这几大类型的调试器分别进行讨论。

1. 独立型

绝大多数常见的调试器都属于独立型调试器。这些调试器作为独立的进程运行，你可以把它们与一个在编译时包括了相应的调试信息（这是为了把二进制代码与源代码对应起来，尤其是代码里的各种符号）的系统关联起来并对之进行调控。除非你正在调试的程序是以源代码形式存在的（比如那些解释型编程语言），否则通常都需要把源代码文件准备在手边才能把调试器进程与被调试的进程关联起来。

在你把调试器进程与被调试的系统或进程关联起来后，就可以通过独立型调试器去调控（停止、启动、单步调试等）被调试的系统或进程了。单步调试包括三种基本的操作。

- (1) 执行当前代码行，然后单步执行下一行代码。
- (2) 忽略下一行代码（执行函数调用，然后返回到下一行）。
- (3) 连续执行多个代码行，直到遇见程序员在某个特定的代码行上事先设置的断点为止。

断点又分为两种情况：一是程序员事先指定的某一行代码；二是调试光标所在的代码行，也叫作“执行到光标位置”。

独立型调试器提供的工具包括内存检测工具、调用栈，甚至有时还有堆。变量检测功能或许是调试器所提供的最重要的诊断工具了。总而言之，只要是由被调试进程保存到某个地方的信息，调试器几乎都可以检测得到。

注解 堆 (heap) 是一种用来保存内存地址的树结构，它可以加快内存块的分配和回收操作。栈 (stack) 是一种“先进-后出”的数据结构，最后进入栈的数据将被最先取出。

独立型调试器的另一个特征是它们通常不是某个开发环境的集成组件之一。换句话说，它们不是编译工具包的组成部分，它们的操作独立于开发环境。使用独立型调试器的好处是有许多在功能上略有差异的调试器可供选用，你可以根据自己的具体情况选择一个最适当的调试器来完成调试工作。

这类调试器当中的典型代表是GNU Debugger (gdb)，你们可以在www.gnu.org/software/gdb/documentation上找到更多关于它的信息。gdb调试器运行在Linux平台上，它可以对在调试模式下编译出来的系统进行控制和调查。代码清单5-5是我编写的一个用来计算阶乘的示例程序。经验丰富的程序员大概一眼就能看出它里面的逻辑错误，但我们现在不妨假设这个程序就是这样运行的。于是，当我输入一个3时，它计算出来的结果是18，而正确的结果应该是6。

代码清单5-5 示例程序 (sample.c)

```
#include <stdio.h>
#include <stdlib.h>

static int factorial(int num)
{
    int i;
    int fact = num;

    for (i = 1; i < num; i++)
    {
        fact += fact * i;
    }
    return fact;
}

int main(int argc, char *argv[])
{
    int num;
```

```

int fact = 0;

num = atoi(argv[1]);
fact = factorial(num);
printf("%d! = %d\n", num, fact);
return 0;
}

```

如果我想用gdb来调试这个程序，必须先用下面这条命令在调试模型下编译它。

```
gcc -g -o sample sample.c
```

完成编译之后，再用下面这条命令启动gdb调试器。

```
gdb sample
```

gdb调试器的命令提示符出现后，我先用break命令（加上源代码文件名和断点位置的行号）设置了一个断点，然后开始运行这个程序，并以命令行参数的形式输入一个3。还可以用print命令打印出任何一个变量的值。如果我想继续执行，可以输入continue命令。最后，在完成调试任务后，可以用quit命令退出gdb调试器。读者可以在代码清单5-6给出的调试会话示例里看到这些命令的用法。

代码清单5-6 用gdb调试器调试一个程序

```
# gdb sample
```

```
GNU gdb 6.3
```

```
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain
conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

```
This GDB was configured as "i586-suse-linux"...Using host libthread_db
library "/lib/tls/libthread_db.so.1".
```

```
(gdb) break sample.c:10
```

```
Breakpoint 1 at 0x804841d: file sample.c, line 10.
```

```
(gdb) run 3
```

```
Starting program: /home/Chuck/source/testddd/sample 3
```

```
Breakpoint 1, factorial (num=3) at sample.c:11
```

```
11      fact += fact * i;
```

```
(gdb) print i
```

```
$1 = 1
```

```
(gdb) print num
```

```
$2 = 3
```

```
(gdb) print fact
```

```
$3 = 3
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 1, factorial (num=3) at sample.c:11
```



```

11      fact += fact * i;
(gdb) continue
Continuing.
3! = 18

Program exited normally.
(gdb) quit
#

```

你看出这个示例程序里的逻辑错误了吗？我来提示一下：为数字3计算阶乘时的第一个值应该是什么？请注意factorial()方法的变量声明部分，尤其是那条fact = num; 声明。

注解 有些人把像gdb这样的调试器称为交互式调试器，因为用户需要以交互方式通过它们与被调试的系统打交道。这种说法有几分道理，但不要忘记gdb是在外部控制着被调试系统的，你只能通过几个非常初级的方法（比如list命令，用来列出源代码）与源代码打交道。若是gdb提供了一个图形化的用户界面来显示源代码，且允许你查看数据和与源代码打交道的話，它才算得上是一个交互式调试器。ddd调试器就是这样做的。

2. 交互式调试器

有些调试器本身属于开发环境的一部分，它们或者是“编译-链接-运行”工具包里的一部分，或者是交互式开发环境的一个集成部分。与独立型调试器不同，交互式调试器使用的是与开发工具相同或非常相似的操作界面。Microsoft Visual Studio .NET中的调试功能就是一个集成化交互式调试器的典型例子。在Visual Studio里，交互式调试器只是快速应用开发过程的一个不同模式而已。创建一个表单，编写一些代码，然后在调试模式下就可以运行它。

图5-3演示了在Visual Studio 2005里调试前面那个示例程序的Windows变体的情况。

交互式调试器与独立型调试器有着同样的功能集。你可以停止、启动、单步运行（分三种情况；见上一小节）被调试的程序。交互式调试最有用的地方是：当你找到故障根源的时候，可以停止被调试程序的执行、做出必要的修改、然后再次运行被调试的程序。表5-1对这些命令进行了简单的描述。绝大多数调试器都支持这些命令（有些甚至支持更多的命令），但它们给这些命令起的名字不尽相同。这些命令的准确名字可以在调试器的文档里查到。

表5-1 基本的调试器控制命令

命 令	说 明
Start (Run)	执行被测试的程序
Stop (Break)	暂时停止代码的执行
Step Into	执行当前语句；把调试光标（焦点）移动到下一条语句。如果被执行的语句是一个函数，调试光标将移动到该函数中的第一条可执行语句上
Step Over	执行当前语句；把调试光标（焦点）移动到下一条语句。如果被执行的语句是一个函数，调试光标将移动到从这个函数调用返回后的下一条可执行语句上
Breakpoint	调试器将在到达断点所在的代码行时停止被调试程序的执行。许多调试器允许你使用条件断点，被调试的程序在断点处是否停止执行，将由一个表达式来决定
Run to Cursor	被调试程序将一直执行到调试光标所在的代码行。从效果上看，这相当于一种一次性的断点

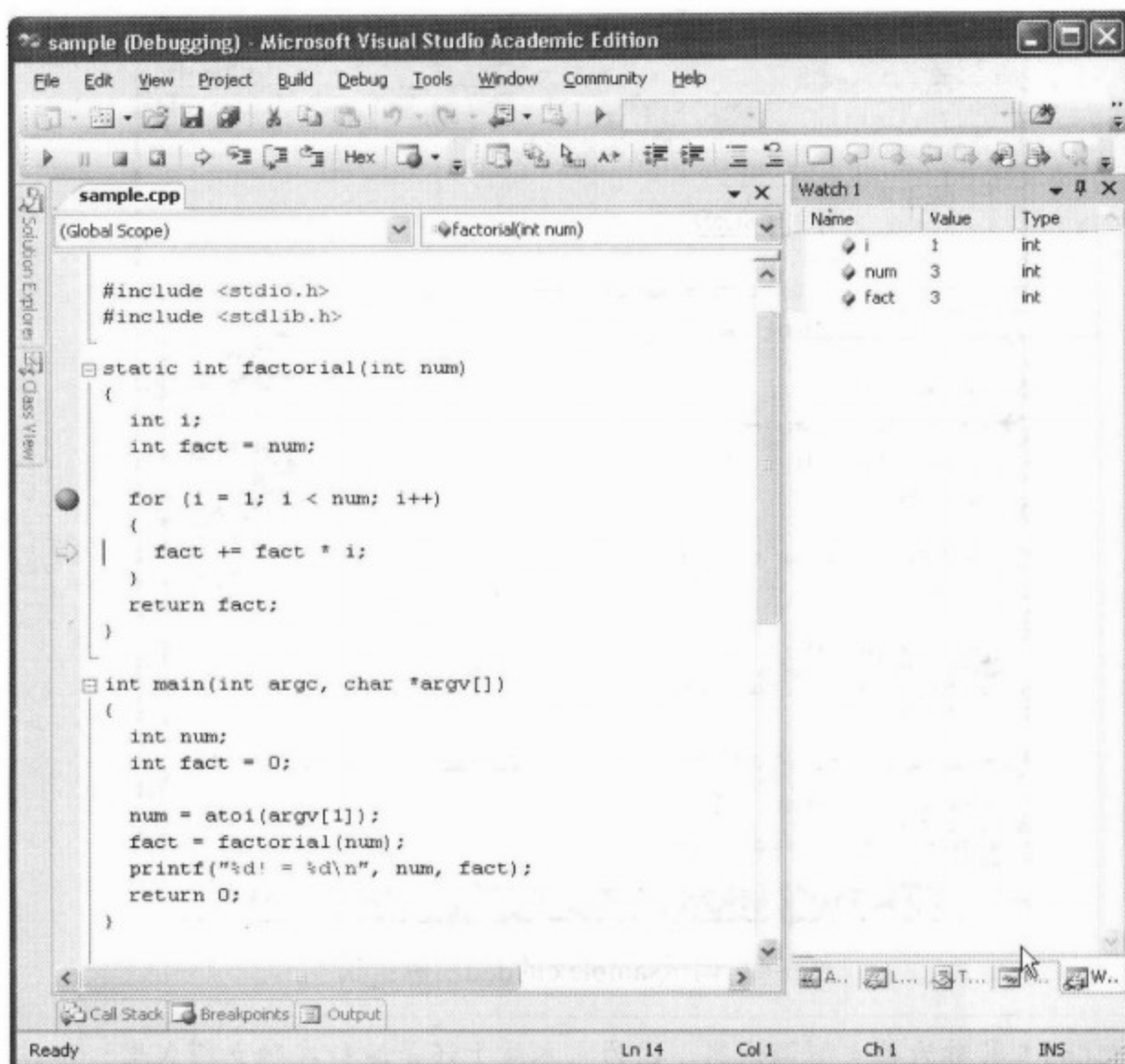


图5-3 Visual Studio调试示例 (sample.c)

在Visual Studio里调试程序的时候，对源代码的编译和链接操作都发生在幕后，你只需稍等片刻就可以回到调试器里。不难想象，交互式调试器可以节省大量的时间。如果你从没用过独立型调试器，可能会对独立型调试器明显缺乏与源代码项目的集成而感到失望，但那些现在看来很“古老”的独立型调试器，一直是绝大多数程序员手里的调试工具，交互式调试器只是在最近几年，才因为快速应用开发工具的进化而成为人们喜欢的调试工具。

3. GNU数据显示调试器

交互式调试器的另一个典型代表是ddd (GNU Data Display Debugger, GNU数据显示调试器)，你们可以在<http://www.gnu.org/software/ddd>网页找到它的下载链接。ddd调试器允许你运行程序并同时看到它的源代码。它在概念上与诸如Visual Studio这样的快速应用开发调试器很相似。图5-4是我们的示例程序在ddd调试器里运行时的情况。

请注意显示在窗口上半部分的变量。在使用ddd调试器的时候，我可以通过用鼠标单击某行代码的办法来设置断点，不必再像过去那样必须记住它在源代码文件里的行编号。我还可以通过双击某个变量的办法来查看它的内容。甚至可以用类似的办法修改一个或多个变量的值，这使我可以用不同的值来试验相关代码的工作情况。这是一个非常有用的功能，它可以让我轻而易举地发现“加减1”错误（例如把应该从0开始计数的循环控制变量错写成了从1开始）。

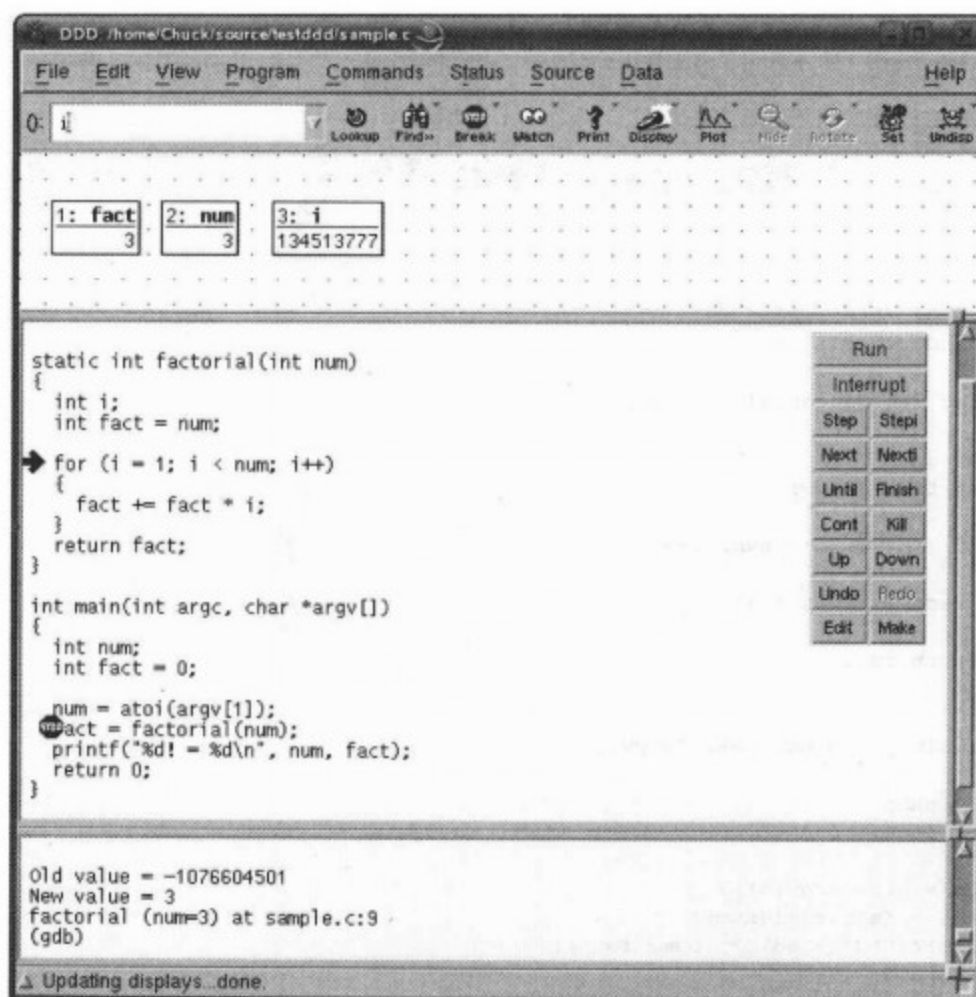


图5-4 调试sample.c的ddd示例会话

注解 有些人把ddd工具称为独立型调试器，因为它本质上还是运行在独立模式里。不过，因为它的图形化用户界面有着与软件开发工具相类似的外观，使我认为ddd调试器更像是一个交互式调试器。不管怎么说，自从ddd调试器出现以后，gdb调试器再也不想以前那么风光了！

4. 双向调试器

现在的调试器功能已经很强大了，但人们仍在研究怎样进一步提高调试工作的效率。最让人感兴趣的是，有些研究人员正在想办法让调试器既能执行操作、也能撤销操作，其目的是为了能够让程序员能够观察到每一个操作的影响，从而发现问题的根源。那些研究人员把这种技术称为逆向推理(backwards reasoning)。他们认为，寻找问题根源最有效的办法是观察代码的执行情况，在故障现象发生时立刻撤销上一步操作，并观察都有哪些东西发生了变化。实现了这种技术的工具被称为双向调试器(bidirectional debugger)。

Undo公司(<http://undo-software.com>)推出的UndoDB是一个商业化的双向调试器产品。UndoDB目前只能在Linux平台上使用，职业程序员需要支付合理的许可证费用，非职业程序员可以免费使用它。虽然UndoDB不是一个开源产品，但Undo公司出于感谢开源阵营的考虑，决定把它们的产品免费提供给那些不靠编写软件谋取报酬和那些不将其产品用于商业目的的人们。

UndoDB是一个使用gdb信息的独立型调试器。与gdb调试器不同的是，UndoDB还提供了一些能够让调试人员把被调试程序的执行方向掉转过来并撤销最后一条语句的命令。代码清单5-7给出了一个用UndoDB来调试示例程序的例子。

代码清单5-7 用UndoDB调试器调试一个程序 (sample.c)

```
# undodb-gdb sample
```

```
Undodb-gdb bi-directional debugging system. Copyright 2006 Undo Ltd.
```

```
undodb-gdb: starting gdb...
```

```
GNU gdb 6.3
```

```
Copyright 2004 Free Software Foundation, Inc.
```

```
GDB is free software, covered by the GNU General Public License, and you are  
welcome to change it and/or distribute copies of it under certain conditions.
```

```
Type "show copying" to see the conditions.
```

```
There is absolutely no warranty for GDB. Type "show warranty" for details.
```

```
This GDB was configured as "i586-suse-linux"...
```

```
Using host libthread_db library "/lib/tls/libthread_db.so.1".
```

```
(gdb) break sample.c:9
```

```
Breakpoint 1 at 0x8048414: file sample.c, line 9.
```

```
(gdb) run 3
```

```
Starting program: /home/Chuck/source/testddd/sample 3
```

```
Breakpoint 1, factorial (num=3) at sample.c:9
```

```
9 for (i = 1; i < num; i++)
```

```
(gdb) next
```

```
11 fact += fact * i;
```

```
(gdb) bnext
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
```

```
0x08048436 in factorial (num=3) at sample.c:9
```

```
9 for (i = 1; i < num; i++)
```

```
(gdb) next
```

```
11 fact += fact * i;
```

```
(gdb) break sample.c:13
```

```
Breakpoint 2 at 0x8048438: file sample.c, line 13.
```

```
(gdb) continue
```

```
Continuing.
```

```
Breakpoint 2, factorial (num=3) at sample.c:13
```

```
13 return fact;
```

```
(gdb) print fact
```

```
$1 = 18
```

```
(gdb) bnext
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
```

```
0x08048436 in factorial (num=3) at sample.c:9
```

```
9 for (i = 1; i < num; i++)
```

```
(gdb) print fact
```

```
$2 = 18
```

```
(gdb) bnext
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.
```

```
0x08048429 in factorial (num=3) at sample.c:11
11 fact += fact * i;
(gdb) print fact
$3 = 6
(gdb) print i
$4 = 2
(gdb) next
9 for (i = 1; i < num; i++)
(gdb) print i
$5 = 2
(gdb) print fact
$6 = 18
(gdb) print num
$7 = 3
(gdb) next
Breakpoint 2, factorial (num=3) at sample.c:13
13 return fact;
(gdb) continue
Continuing.
3! = 18

(gdb) quit
The program is running. Exit anyway? (y or n) y
#
```

请注意代码清单5-7里的**bnext**命令。**bnext**是UndoDB调试器独有的命令之一，它可以逆向跟踪被调试程序的执行情况。UndoDB调试器里的所有逆向跟踪命令都对应于某个gdb命令，这使得那些熟悉gdb的程序员可以迅速掌握这个调试器的用法。

殊途同归

你们可能会对我着重介绍那些“古老”的调试技术，而对眼下非常时髦的交互式开发潮流关注不多感到奇怪。我承认，有些调试方法在特定的环境下甚至是在更大的范围内会比另一种更好，但我在这里介绍的方法（以及很多来不及介绍的方法）都可以带来成功的结果也是事实。软件公司不应该硬性规定本公司的程序员必须使用某种特定的工具或方法（这个问题并不仅限于调试工作），因为对某一个人或某一件事有好处的东西，不见得对另一个人或另一件事也有好处。我的建议是，应该根据你的具体需要和项目选择最好的调试工具或方法。选用什么方法并不重要，只要它能让你高效率地调试好项目就行。只要你有良好的故障排除技能并能够获得发现问题所需要的信息，怎么解决问题并不重要。

5.3 调试 MySQL

你们可能很擅长调试自己的应用程序，它们当中肯定也有一些大块头。可是，有机会去调试一个像MySQL这样的大系统的人可不多。虽然谈不上有多困难，但我在研究MySQL源代码的时候还是遇

到了许多挑战。希望接下来的几个小节能够把我经过反复尝试才获得的知识传授给大家。建议你们至少把这部分内容通读一遍，并在空闲时间练习一下这里给出的例子。

我将从一个利用内嵌调试语句调试一段MySQL代码的例子开始本节的讨论。稍后还会向大家提供一个出错处理器的例子，并对如何在Linux和Windows平台上调试MySQL做进一步的介绍。如果你一直在等待机会深入MySQL源代码的内部，那么本小节就是为你准备的。卷起袖子，喝点儿咖啡，开始工作！

5.3.1 内嵌调试语句

MySQL AB公司向它们的顾客提供了一个健壮的内嵌调试语句调试工具，这个工具脱胎于Fred Fish首创的调试器，MySQL AB公司的创始人之一Michael Widenius后来又给它增加了一些线程安全方面的功能。这个工具其实是一系列用C语言编写的宏命令，人们把它们统称为DEBUG。

DEBUG很容易使用，只要在你想把什么东西记录下来的地方插入一条简单的代码语句就行了。MySQL AB公司的程序员在MySQL的源代码里给我们留下许多非常好的例子，它们记录了服务器执行情况的许多方面。MySQL的随机文档把这些宏命令称为DEBUG标记。目前用在MySQL源代码里的DEBUG标记包括以下这些：

- ❑ DEBUG_ENTER，用函数的定义来表明进入了函数。
- ❑ DEBUG_EXIT，记录函数的返回结果。
- ❑ DEBUG_INFO，记录诊断信息。
- ❑ DEBUG_WARNING，记录不常见的事件或没有预料到的事件。
- ❑ DEBUG_ERROR，记录出错代码（主要用在出错处理器里）。
- ❑ DEBUG_LOOP，进入或退出循环。
- ❑ DEBUG_TRANS，记录事务信息。
- ❑ DEBUG_QUIT，记录导致系统非正常关闭的错误。
- ❑ DEBUG_QUERY，记录查询语句。
- ❑ DEBUG_ASSERT，记录某个表达式未能通过测试的原因。

代码清单5-8给出了部分DEBUG标记在mysql_show_privileges()函数里的用法。以黑体字突出显示的代码语句是一些比较常见的DEBUG标记。

代码清单5-8 DEBUG标记的用法示例

```
bool mysql_show_privileges(THD *thd)
{
    List<Item> field_list;
    Protocol *protocol= thd->protocol;
    DEBUG_ENTER("mysql_show_privileges");

    field_list.push_back(new Item_empty_string("Privilege",10));
    field_list.push_back(new Item_empty_string("Context",15));
    field_list.push_back(new Item_empty_string("Comment",NAME_LEN));
    if (protocol->send_fields(&field_list,
        Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
```



```

        DEBUG_RETURN(TRUE);

show_privileges_st *privilege= sys_privileges;
for (privilege= sys_privileges; privilege->privilege ; privilege++)
{
    protocol->prepare_for_resend();
    protocol->store(privilege->privilege, system_charset_info);
    protocol->store(privilege->context, system_charset_info);
    protocol->store(privilege->comment, system_charset_info);
    if (protocol->write())
        DEBUG_RETURN(TRUE);
}
send_eof(thd);
DEBUG_RETURN(FALSE);
}

```

如果把DEBUG标记列成表格的话，那会是一份相当长的清单。DEBUG_ENTER和DEBUG_RETURN标记是最有用的，它们可以把程序执行过程中曾经调用过的所有函数记录到踪迹文件里。我想特别告诉大家的是，MySQL源代码里的每一个函数在其入口点和出口点都分别加有这些标记。如果你想把你编写的函数添加到MySQL源代码树里去，就应该在函数的入口点和出口点也加上这些标记。在Linux平台上，这些标记将把有关信息写入踪迹文件/tmp/mysql.trace；在Windows平台上的踪迹文件是c:\mysql.trace。

请注意，踪迹文件可能会变得非常大。还好，你可以通过命令行参数来控制哪些标记可以把信息写入踪迹文件。比如说，如果你想让踪迹文件只包含一些比较有意思的DEBUG标记，可以使用如下所示的命令。打开DEBUG开关a、b和c的通用格式是a:b:c。带参数的开关必须用逗号隔开。

```

mysql-debug --debug=d,info,error,query,general,where:t:l:g:0,
/tmp/mysql.trace -u root

```

上面这条命令运行debug模式下编译出来的MySQL服务器（mysql-debug）。命令行参数--debug=d,info,error,query,general,where:t:l:g:0,/tmp/mysql.trace告诉DEBUG系统：启用嵌在MySQL源代码中的DEBUG_INFO、DEBUG_ERROR、DEBUG_QUERY和DEBUG_WHERE标记，启用各函数入口点和出口点处的DEBUG_ENTER和DEBUG_RETURN标记，记录这些调试语句在源代码里的行号，启用监控功能，把信息写入踪迹文件/tmp/mysql.trace。-u root参数将把用户名root传递给服务器。还有许多其他的选项，表5-2^①列出了一些比较常用的选项。

表5-2 常用的DEBUG开关

开 关	说 明
d	启用随后列出的DEBUG标记。如果没有列出任何标记，则启用所有的标记
D	在每次输出后加上一段延时。这个参数的值以十分之一秒为单位。比如说，“D.40”将导致4秒的延时
f	只允许d参数给出的清单里的调试、跟踪和监控标记向踪迹文件写信息

① 常用DEBUG开关的完整清单可以在《MySQL参考手册》附录E中的E.3。

(续)

开 关	说 明
F	给踪迹文件的每一行内容加上相应的源代码文件的名字
I	给踪迹文件的每一行内容加上相应的进程ID或线程ID
g	启用监控功能。你可以在这个参数的后面把想要监控的关键字列出来。如果没有列出任何关键字，则监控所有的关键字
L	给踪迹文件的每一行内容加上相应的源代码行号
n	设置每个输出行的嵌套深度。这可以让踪迹文件的内容更容易阅读
N	给踪迹文件的每一行内容加上一个序号
o	把输出保存到这个参数所指定的文件。该参数的默认设置值是stderr
O	把输出保存到这个参数所指定的文件。该参数的默认设置值是stderr。每次调试前先删除该文件里的现有内容
P	给踪迹文件的每一行内容加上当前进程的名字
t	打开函数调用/退出跟踪线（用垂线字符 来表示）

代码清单5-9节选自执行show authors;命令后得到的踪迹文件。可以看到MySQL系统执行这条命令和返回数据时的完整过程（删去了许多行，因为这份清单是由默认的DEBUG开关生成的）。还请注意输出行前面的跟踪行（垂线字符“|”），它们可以帮助你跟踪代码的执行顺序。

如果你想把编写的函数添加到MySQL里去，可以用DEBUG标记把自己的信息记录到踪迹文件里。当你的代码引起一些难以预料的行为时，这个文件可以帮你尽快查明问题的原因。

代码清单5-9 踪迹文件示例（Show Privileges命令）

```
338: | | | >mysqld_show_privileges
```

```
171: | | | >alloc_root
220: | | | <alloc_root
171: | | | >alloc_root
220: | | | <alloc_root
171: | | | >alloc_root
220: | | | <alloc_root
171: | | | >alloc_root
220: | | | <alloc_root
171: | | | >alloc_root
220: | | | <alloc_root
171: | | | >alloc_root
220: | | | <alloc_root
550: | | | >send_fields
171: | | | >alloc_root
220: | | | <alloc_root
127: | | | >_mymalloc
202: | | | <_mymalloc
261: | | | >_myfree
315: | | | <_myfree
127: | | | >_mymalloc
```


请注意这份清单末尾部分显示的数据。在诊断与时序、内存页面、锁定和上下文切换等方面有关的问题时，这些统计数据往往能帮上大忙。在遇到麻烦的时候，到代码映像文件和踪迹文件里查查总是会有所收获的。

5.3.2 出错处理器

出错处理器是个好东西，但让我在MySQL里找个工具来证明这一点，还真有点儿困难。这里只想提醒大家一句：一定要在你编写的出错处理器里对所有可能发生的错误做出妥善的处理。下面列举了一个没能做到这一点的出错处理器，希望这个反面教材能让读者了解怎样做才是最好的。代码清单5-10节选自适用于Windows平台的MySQL 5.0.15版源代码，这段代码在遇到一个特定类型的错误时会出问题。

代码清单5-10 MySQL中的错误处理程序示例

```
int my_delete(const char *name, myf MyFlags)
{
    int err;
    DEBUG_ENTER("my_delete");
    DEBUG_PRINT("my", ("name %s MyFlags %d", name, MyFlags));

    if ((err = unlink(name)) == -1)
    {
        my_errno=errno;
        if (MyFlags & (MY_FAE+MY_WME))
            my_error(EE_DELETE, MYF(ME_BELL+ME_WAITTANG+(MyFlags & ME_NOINPUT)),
                    name, errno);
    }
    DEBUG_RETURN(err);
} /* my_delete */
```

看出问题了吗？还是我来告诉你吧。在Windows环境里，unlink()函数的返回值有几个重要的值需要检查，但代码清单5-10里的出错处理器没有对其中某个值进行检查。这个缺陷将导致optimize()函数在它执行期间错误地复制一个中间文件。不过，这个缺陷在你们看到本书时应该已经被修补好了。

MySQL AB公司提供了一个设计优良的出错消息机制，它可以让你们的出错处理器更加健壮。如果你想添加自己的出错消息，可以把它们添加到sql/errmsg.txt文件里。添加你自己的出错消息的具体细节可以在internals.pdf文档里找到。

在编写一个出错处理器的时候，一定要把所有可能发生的错误考虑周全并采取适当的行动来纠正或者报告它们，这句话怎么强调也不过分。用DEBUG标记来追踪和记录出错消息，可以让你们的调试工作更有效率。

5.3.3 在Linux环境里调试MySQL

高质量的开发工具（尤其是各种GNU工具）是Linux取得巨大成功的原因之一。就拿调试器来说吧，Linux平台上的许多调试器不仅可以用来调试单线程系统，还可以用来调试多线程系统。

可以在Linux平台上使用的调试器很多，其中最受欢迎的是gdb和ddd。在接下来的几个小节里，我

为这两种调试工具各自准备了一个调试MySQL系统的例子，这两个例子的场景是分析发出SHOW AUTHORS命令时会发生什么情况。下面先从gdb调试器开始讨论，稍后再向大家介绍ddd调试器的用法。

1. 使用gdb调试器

请大家先回到代码清单5-1去重温一下show_authors()函数的完整代码。需要做的第一件事情是在调试模式下编译我的服务器。具体地说，就是在MySQL源代码的根文件夹里发出如下所示的命令：

```
./configure --with-debug
make
make install
```

这些命令将把相应的调试信息添加到编译出来的MySQL系统里，这样才能用调试器去调试它。接下来，我将使用mysqld-debug命令启动这个服务器。代码清单5-11是服务器启动时给出的启动信息。

注意 你应该先关闭所有正在运行的MySQL服务器，再启动你在调试模式下编译出来的那个服务器。也可以不这样做——这只是以防万一，以避免费了半天劲才发现自己调试的是另外一个MySQL服务器进程。

代码清单5-11 在调试模式下启动MySQL服务器

```
linux:~ # mysqld-debug -uroot
060530 20:42:07 InnoDB: Started; log sequence number 0 46403
060530 20:42:07 [Note] mysqld-debug: ready for connections.
Version: '5.1.9-beta-debug' socket: '/var/lib/mysql/mysql.sock' port: 3306
MySQL Community Server - Debug (GPL)
```

请注意，我正在使用的是/var/lib/mysql/mysql.sock套接字。这可以让我在调试模式下运行MySQL服务器的一份副本，而不会影响到另一个正在运行的MySQL服务器。不过，还需要让客户程序使用同一个套接字才行。但首先，需要确定我这个服务器的进程ID。可以用ps -A命令列出所有正在运行的进程并找出这个ID。另外一个办法是用ps -A | grep mysql命令只列出名字里带有mysql字样的进程ID。下面是这个命令的执行结果：

```
9740 pts/2    00:00:00 mysqld
```

查出MySQL服务器的进程ID之后，就可以启动gdb调试器并用attach 10592命令把它关联到正确的进程上了。我还想在show_authors()函数里设置一个断点。在仔细查看过有关的源代码文件后，我决定把断点设在第207行，相应的命令是break /home/Chuck/MySQL/mysql-5.1.9-beta/sql/sql_show.cc:207。这个命令的格式是file:line#。设置好了断点，就可以发出continue命令让那个进程开始执行了，gdb调试器将在遇到断点的时候让程序暂停下来。代码清单5-12给出了一个完整的调试过程。

代码清单5-12 运行gdb调试器

```
# gdb
```

```
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
```

GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details.

This GDB was configured as "i586-suse-linux".

(gdb) attach 10592

Attaching to process 10592

warning: could not load vsyscall page because no executable was specified

warning: try using the "file" command first

Reading symbols from /usr/sbin/mysqld-debug...done.

Using host libthread_db library "/lib/tls/libthread_db.so.1".

Reading symbols from /lib/tls/libpthread.so.0...done.

[Thread debugging using libthread_db enabled]

[New Thread 1075779264 (LWP 10592)]

[New Thread 1098349488 (LWP 10636)]

[New Thread 1098148784 (LWP 10601)]

[New Thread 1106926512 (LWP 10600)]

[New Thread 1104825264 (LWP 10599)]

[New Thread 1102724016 (LWP 10598)]

[New Thread 1095846832 (LWP 10596)]

[New Thread 1093745584 (LWP 10595)]

[New Thread 1091644336 (LWP 10594)]

[New Thread 1089543088 (LWP 10593)]

Loaded symbols for /lib/tls/libpthread.so.0

Reading symbols from /lib/tls/libc.so.6...done.

Loaded symbols for /lib/tls/libc.so.6

Reading symbols from /lib/libnss_files.so.2...done.

Loaded symbols for /lib/libnss_files.so.2

Reading symbols from /lib/libnss_dns.so.2...done.

Loaded symbols for /lib/libnss_dns.so.2

Reading symbols from /lib/libresolv.so.2...done.

Loaded symbols for /lib/libresolv.so.2

Reading symbols from /lib/libcrypt.so.1...done.

Loaded symbols for /lib/libcrypt.so.1

Reading symbols from /lib/libnsl.so.1...done.

Loaded symbols for /lib/libnsl.so.1

Reading symbols from /lib/tls/libm.so.6...done.

Loaded symbols for /lib/tls/libm.so.6

Reading symbols from /lib/ld-linux.so.2...done.

Loaded symbols for /lib/ld-linux.so.2

Reading symbols from /lib/libgcc_s.so.1...done.

Loaded symbols for /lib/libgcc_s.so.1

0xffffe410 in ?? ()

(gdb) break /home/Chuck/MySQL/mysql-5.1.9-beta/sql/sql_show.cc:207

Breakpoint 1 at 0x82e32bc: file sql_show.cc, line 207.

(gdb) continue

Continuing.


```

~ [Switching to Thread 1098349488 (LWP 10636)]

Breakpoint 1, mysqld_show_authors (thd=0x8f30100) at sql_show.cc:207
207      field_list.push_back(new Item_empty_string("Name",40));
(gdb) next
208      field_list.push_back(new Item_empty_string("Location",40));
(gdb) next
209      field_list.push_back(new Item_empty_string("Comment",80));
(gdb) next
212                                     Protocol::SEND_NUM_ROWS |
Protocol::SEND_EOF))
(gdb) next
216      for (authors= show_table_authors; authors->name; authors++)
(gdb) next
218      protocol->prepare_for_resend();
(gdb) print authors->name
$1 = 0x877ac9f "Brian (Krow) Aker"
(gdb) quit

```

为了看到服务器的动作，还需要在运行gdb调试器的同时启动一个客户程序来发出查询命令。下面这条命令启动了MySQL命令行客户端程序。

```
mysql -u root -p -S /var/lib/mysql/mysql.sock
```

代码清单5-13是客户端程序给出的初始化信息，它将使用命令行中指定的套接字。接着发出SHOW AUTHORS命令。

代码清单5-13 启动MySQL客户端程序去连接服务器

```

Chuck@linux:~> mysql -u root -p -S /var/lib/mysql/mysql.sock
Enter password:

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.1.9-beta-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show authors;

```

当我输入SHOW AUTHORS命令的时候，注意到的第一件事是客户端程序没有反应了。这是因为gdb调试器已经遇到了刚才设置的那个断点并暂停了程序的执行。现在，可以切换到调试器去发出调试命令了。可以用next命令单步执行，也可以用print命令显示变量的值（参见代码清单5-12）。在完成这次调试任务之后，就可以关闭服务器并退出调试器了。

gdb调试器是一个功能强大的工具，但用起来不像大多数集成化开发环境（integrated development environment, IDE）里的调试器那么方便。ddd调试器填补了这个空白，它向用户提供了一个健壮的图形化操作环境。

2. 使用ddd调试器

GNU推出的ddd调试器是集成化调试器的一个典型例子。尽管不是专为某个IDE环境而开发的，ddd

调试器还是可以让用户获得类似的体验。你可以启动想调试的程序并查看它的源代码。在使用这个集成化调试工具的时候，可以设置断点，暂停和开始被调试程序的执行，查看变量的值，查看栈的变化情况，甚至可以修改变量的值。

ddd调试器有好几个窗口。它的数据窗口显示着你想查看的所有数据项，源代码窗口（主显示区）显示着被调试程序的当前源代码，调试器控制台窗口显示着底层调试器（gdb）的输出。你既可以通过菜单去控制正在调试的程序，也可以通过调试器控制台窗口直接输入各种调试命令。

ddd调试器的内核是GNU gdb独立型调试器。ddd调试器的开发者在gdb调试器的基础上为它增加了一系列新的功能，这种充分利用别人的成果（gdb）而非另起炉灶的做法，正是开源理念所倡导的。不仅如此，除了gdb调试器，ddd调试器还支持另外几种独立型调试器，这使它成为一个多面手。事实上，它可以支持其底层调试器所支持的任何一种程序设计语言。ddd调试器在许多方面都称得上是一个集成化调试器的典型代表。只要程序是用它能够支持的语言编写的，它就能够提供所需要的调试功能。

我最欣赏的ddd调试器功能之一，是它允许你把尚未完成的调试工作保存起来，等有时间的时候再接着干。这样一来，你就不必每次从头开始重复调试步骤了——只要你曾经调试过一个比较大的程序，就肯定能体会到该功能的优越。我建议大家这样来使用这个功能：当你把程序调试到即将发生故障现象的那一刻（比如说，刚进入你认为有缺陷的那个函数）时，把所有需要观察的变量和需要设置的断点都安排好，然后把这次调试会话保存起来。这样一来，万一接下来的调试出了什么差错，你可以立即回到刚才保存的位置重新开始而不必从头再来。这个办法虽然不如使用一个双向调试器那么有效率，但因此而节省下来的时间往往也相当可观。

还可以使用ddd调试器查看系统崩溃前保存的内存映像文件，这可以让你了解到在系统即将崩溃之前你的程序正处于什么状态，以及它最后执行的几步操作。这个功能非常有用，因为会导致系统崩溃的缺陷往往也会让调试器陷入崩溃^①。ddd调试器还支持远程调试和直接查看内存。换句话说，你可以在开发工作站上通过ddd调试器去调试一个运行在另一台计算机上的程序（那通常是一个服务器）。关于ddd调试器的更多信息可以在www.gnu.org/software/ddd/ddd.html#Doc上的文档里查到。

下面是用ddd调试器去调试MySQL系统时的基本步骤。

- (1) 关闭所有正在运行的MySQL服务器。使用mysqladmin -uroot -p shutdown命令并输入你的root用户密码。
- (2) 进入程序的源代码目录。比如说，如果你想调试的是MySQL服务器（mysqld），需要进入sql子目录。
- (3) 用ddd mysqld-debug命令启动ddd调试器。
- (4) 打开你想调试的源代码文件。在后面的讨论里，我将使用sql_show.cc文件作为例子。
- (5) 把所有的断点设置好。在后面的讨论里，我将在show_authors()函数中的第207行设置一个断点。
- (6) 通过Program ► Run菜单启动服务器运行。如果你想让服务器以root用户的权限运行，别忘了在对话框里给出参数-u root。
- (7) 启动MySQL客户端程序。在后面的讨论里，我将使用最基本的MySQL命令行客户端程序。
- (8) 在客户端程序里发出命令。调试器会在被调试的程序执行到事先安排好的断点位置时让它暂

^① 系统和调试器同时崩溃是最让人头疼，也是最难处理的问题。在遇到这种情况的时候，我的最后绝招是使用内嵌调试语句和内存映像文件进行调试。

停下来，而调试工作就将从这里正式开始。

(9) 在完成调试工作之后，先退出调试器，再使用mysqladmin-uroot-p shutdown命令和你的root用户密码关闭服务器。

提示 你可能需要延长MySQL客户端程序的超时时间。调试需要花费时间，如果你设置的断点较多或者是查看许多变量，花费的时间就更长，而在此期间系统实际上是处于等待状态的。如果等待时间过长，服务器与客户端程序之间的通信就会中断。有些客户端程序会在一定时间内无法与服务器进行通信时自动结束本次会话。如果你使用的是MySQL命令行客户端程序，将需要延长超时时间。可以在命令行上使用--connection-timeout=参数来设定这个值。比如说，--connection-timeout=600将把等待时间设置为600秒，这意味着你在客户端程序放弃连接之前有10分钟的调试时间。

代码清单5-14演示了用ddd调试器去调试MySQL服务器的情况。我在这里选用的例子与前面一样，还是来自sql_show.cc源代码文件的show_authors()函数。但在这个例子里，我感兴趣的是服务器如何把信息发送给客户端程序。读者应该记得，本书在第3章曾经说过，要在本章给出一个分析MySQL服务器是如何把数据返回给客户端程序的例子，下面就是这样一个例子。

代码清单5-14 show_authors()函数（请注意代码中的黑体字部分）

```

/*****
** List all Authors.
** If you can update it, you get to be in it :)
*****/

bool mysqld_show_authors(THD *thd)
{
    List<Item> field_list;
    Protocol *protocol= thd->protocol;
    DEBUG_ENTER("mysqld_show_authors");

    field_list.push_back(new Item_empty_string("Name",40));
    field_list.push_back(new Item_empty_string("Location",40));
    field_list.push_back(new Item_empty_string("Comment",80));

    if (protocol->send_fields(&field_list,
                             Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DEBUG_RETURN(TRUE);

    show_table_authors_st *authors;
    for (authors= show_table_authors; authors->name; authors++)
    {
        protocol->prepare_for_resend();
        protocol->store(authors->name, system_charset_info);
        protocol->store(authors->location, system_charset_info);
        protocol->store(authors->comment, system_charset_info);
    }
}

```



```

        if (protocol->write())
            DEBUG_RETURN(TRUE);
    }
    send_eof(thd);
    DEBUG_RETURN(FALSE);
}

```

以黑体字突出显示的语句就是用来把数据发送给客户端程序的方法。show_authors()函数大概是最适合用来演示这一过程的东西了——没有复杂的操作，只是发送数据而已。以黑体字突出显示的第一条语句声明了一个指针，它指向当前线程的protocol类。protocol类封装了所有低端的通信方法（建立连接、对套接字进行控制等）。接下来的一组语句建立了一个字段列表。MySQL服务器总是先向客户端程序发送一个字段列表。把字段列表构造出来之后，protocol->send_fields()方法将把它发送给客户端程序。接下来是一个循环，它将遍历show_table_authors_st链表并把存放在其中的作者（MySQL的开发人员）名单发送给客户端程序。在这个循环里，把数据发送给客户端程序的具体工作主要通过3个方法完成：第一个是protocol->prepare_for_resend()方法，它负责清理与发送数据有关的缓冲区和变量；第二个是protocol->store()方法，它负责把信息放入发送缓冲区，你必须为你想要发送的每一个字段分别调用一次这个方法；最后一个是protocol->write()方法，它负责完成把数据实际发送给客户端程序的一系列动作。最后，send_eof()函数指示通信机制发出一个eof（end-of-file，文件尾）标志以表明数据已全部发送完毕。在收到这个信号后，客户端程序开始把接收到数据显示给用户。

现在，我们用ddd调试器去看看这个函数是如何工作的。我已经用如下所示的命令，在调试模式下编译好了服务器：

```

./configure --with-debug
make
make install

```

这些命令将把相应的调试信息添加到编译出来的MySQL系统里，这样才能用调试器去调试它。在确认没有其他服务器在运行之后，我启动了ddd调试器，加载了源代码文件（sql_show.cc），在show_authors()函数里设置好断点，把连接等待时间设置为10分钟，然后用MySQL客户端程序发出SHOW AUTHORS命令。服务器的启动情况见前面的代码清单5-12；客户端程序的启动情况见下面的代码清单5-15。

代码清单5-15 启动MySQL客户端程序与ddd调试器配合工作

```

Chuck@linux:~> mysql -u root -p --connection-timeout=600
Enter password:

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 5.1.9-beta-debug

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> show authors;

```

当执行到我在ddd调试器里预先设置的断点时，服务器将暂停下来，ddd调试器的代码窗口将在断点处的语句前显示一个箭头。你还会看到客户端程序没有反应了。请注意，如果调试工作花费的时间

比较长，客户端程序可能会因为等待超时而自动结束运行；这正是我刚才为什么要把连接等待时间设置为10分钟的原因。

调试器让被调试程序暂时停止执行后，你就可以去分析代码和查看变量值、栈和内存了。具体到这个例子，我决定用ddd调试器去查看一下authors结构，看它会把什么样的数据发送给客户端程序。在图5-5给出的ddd调试器工作画面里，可以在它的数据窗口中看到authors结构。



图5-5 用ddd调试器调试show_authors()函数

还可以展开authors结构并查看它的当前内容。在图5-6给出的ddd调试器工作画面里，读者可以在它的数据窗口中看到authors结构的当前内容。

请注意在数据窗口里显示的数据值和地址。ddd调试器还允许你修改内存的内容。比如说，如果在调试这个函数的时候想改变authors结构的内容，只需用鼠标右键单击authors结构里的某个数据项，再从右键弹出菜单里选择Set Value，就可以修改它的值了。图5-7是对authors结构的内容进行修改后的样子。

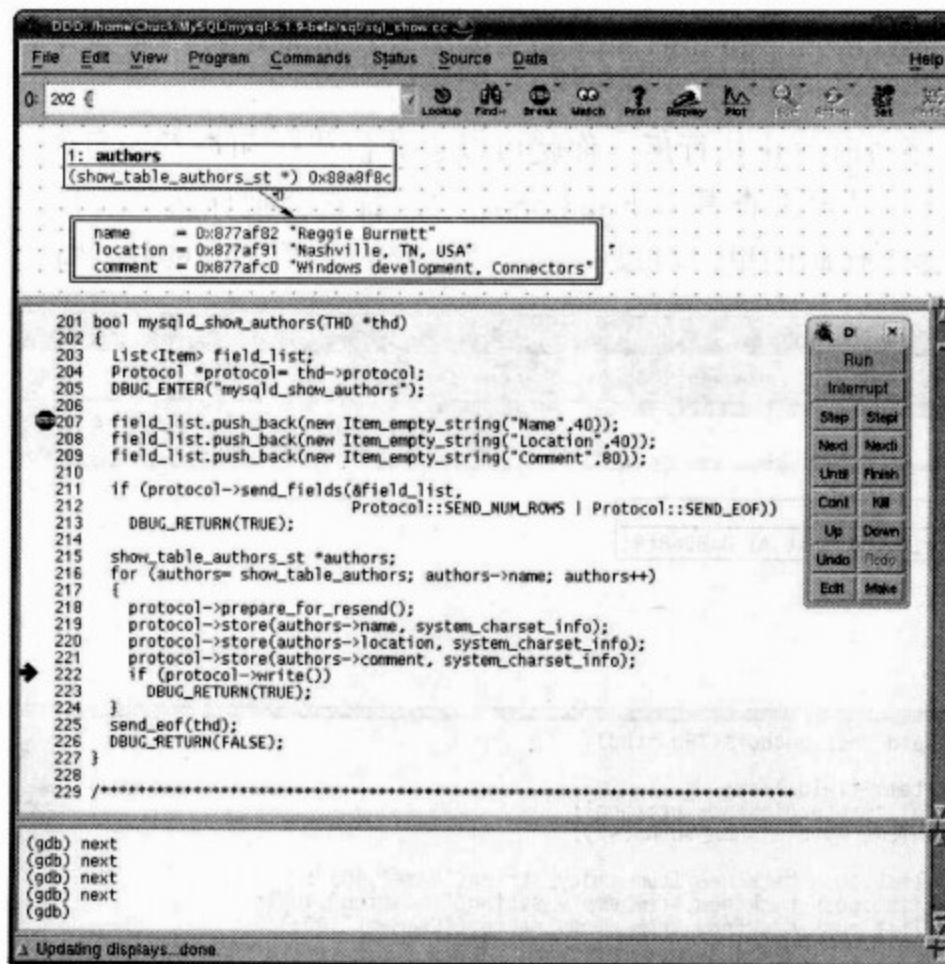


图5-6 用ddd调试器查看authors结构里的数据

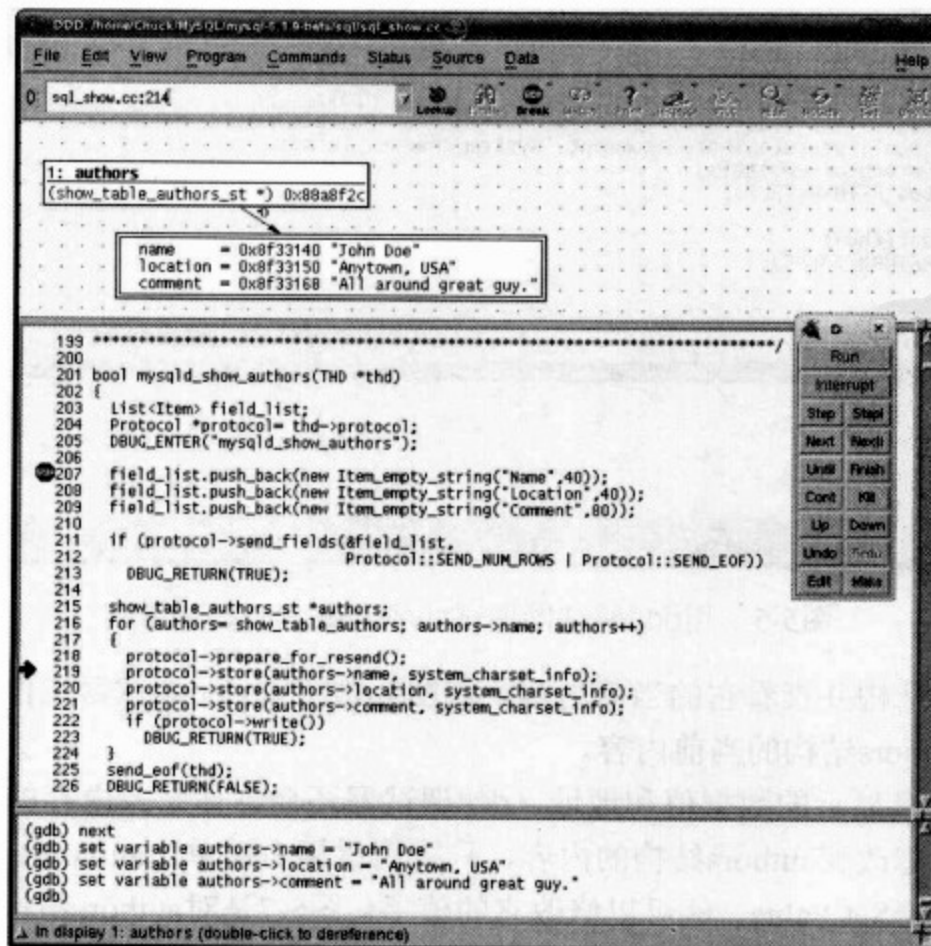


图5-7 用ddd调试器修改authors结构里的数据

读者或许会怀疑这么做是否真的有效果，那就让事实来证明好了。代码清单5-16是MySQL客户端程序的输出（为简洁起见，对它进行了删节）。请看，客户端程序接收到的数据就是我修改出来的东西。

代码清单5-16 用ddd调试器修改authors结构里的数据后的结果输出

```
+-----+-----+-----+
| Name      | Location    | Comment      |
+-----+-----+-----+
| John Doe   | Anytown, USA | All around nice guy.
...
+-----+-----+-----+
74 rows in set (48.35 sec)

mysql>
```

完成这次调试工作后，先退出ddd调试器，然后发出下面这条命令关闭服务器^①。

```
mysqladmin -uroot -p shutdown
```

正如读者在这个例子里看到的那样，ddd调试器用起来非常方便。它不仅可以让你在运行和调试某个程序的同时看到它的源代码，还可以让你实时地对内存里的数据进行查看和修改；这对迅速发现和改正程序代码里的漏洞很重要。应该多找些例子来练习使用ddd调试器，直到你能熟练掌握它为止。

5.3.4 在 Windows 环境里调试 MySQL

在Windows环境里从事调试工作的主要手段是使用Microsoft Visual Studio .NET。有些程序员喜欢使用其他的工具，如外部调试器等，但绝大多数程序员会选用Microsoft Visual Studio .NET里集成的调试器。集成的调试器用起来很方便，因为你可以从同一个操作界面进行编译和调试。

注解 在Windows平台上，老版本的MySQL源代码都带有适用于Microsoft Visual Studio 6或Visual Studio .NET 2003的项目文件和解决方案文件。你可以把这些项目文件和解决方案文件转换为适用于Visual Studio .NET 2005的。本小节里的例子用的都是Visual Studio .NET 2005 Academic Version（学术版）。“学术版”在功能方面没有什么欠缺，所谓学术意思它可以对学生和老师打折销售。许多软件厂商都有类似的销售政策。

我将继续使用上一节介绍ddd调试器时用过的例子。这两种调试器的使用步骤大同小异，但在操作细节方面还是有一些区别的。首先，启动Visual Studio并打开MySQL源代码根目录里的mysql.sln解决方案文件。请注意，这里必须把编译任务设置为在win32平台上以debug模式进行编译；这将确保必要的调试信息会被编译到可执行文件里。在启动Visual Studio并设置好编译模式后，就可以设置断点了（还是设在show_authors()函数的第207行）。图5-8是设好断点之后的Visual Studio工作画面。

① 原书中的操作顺序——先关闭服务器、后退出调试器——是公认的不良习惯。一般原则是：最后打开的程序或文件应该最先关闭。现已改正过来。——译者注

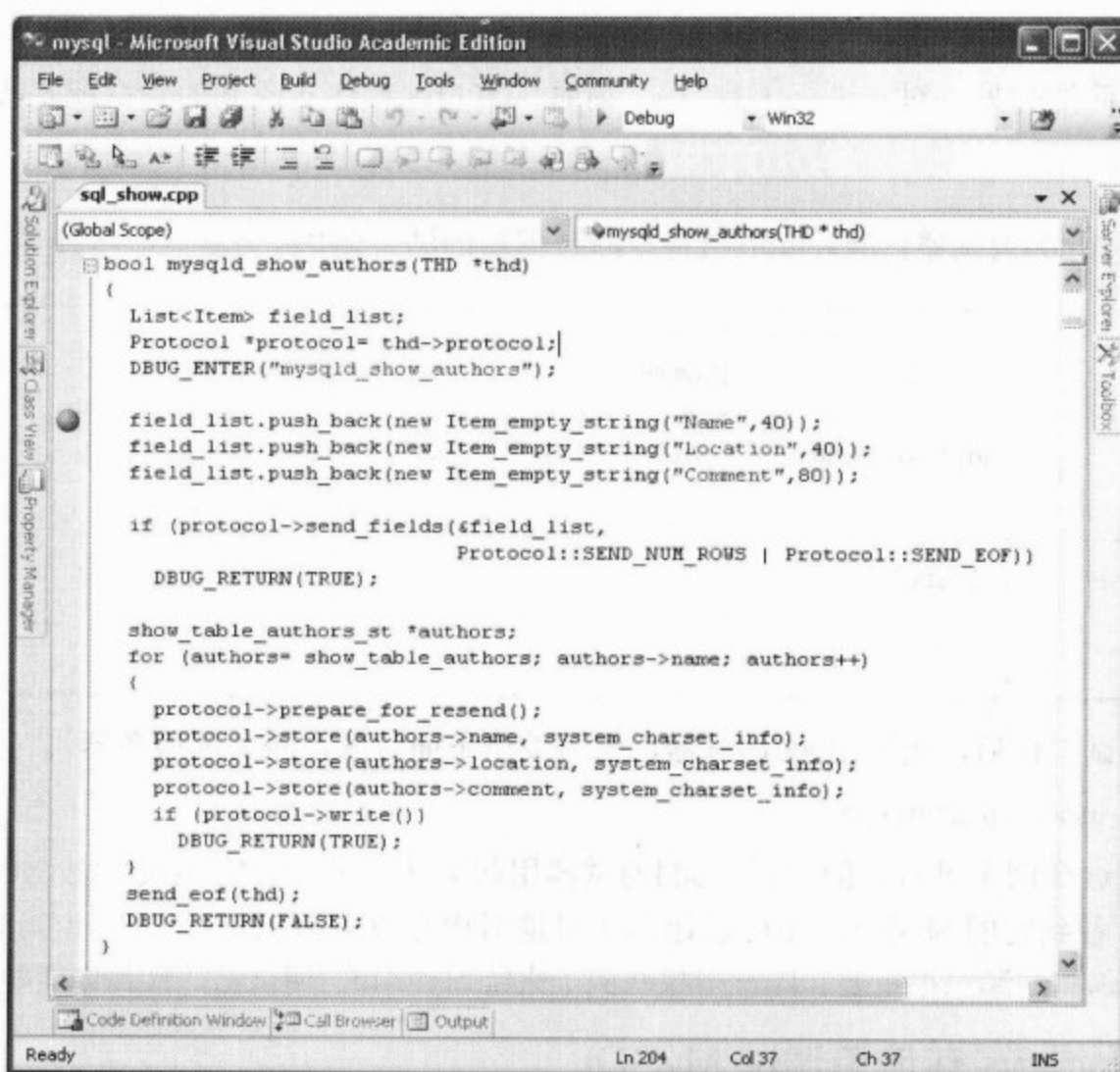


图5-8 Visual Studio调试器的设置窗口

注意 在Visual Studio .NET 2005里编译MySQL源代码的时候，你可能会看到许多过时警告信息。这些编号为C4996的警告信息会告诉你老版本的底层函数已经有了更新的版本。还好，那些老函数仍允许继续使用，编译出来的MySQL服务器也能正确地运行。你可以在项目文件里把这个警告消息关掉。

为了调试MySQL服务器，必须在调试模式下启动它。在Windows平台上，应该使用一个命令行开关单独运行MySQL服务器，而不是作为一项服务运行。你也完全可以不这样做，但这么做的好处是你将能在命令窗口里看到来自MySQL服务器的任何消息，把MySQL服务器运行为一项服务就没有这个好处了。可以用下面这条命令来做到这一点：

```
mysqld-debug --debug --standalone
```

启动MySQL服务器之后，用菜单命令Debug | Attach to Process把它的进程与Visual Studio关联起来。在如图5-9所示的Attach to Process对话框里，运行并关联的是mysqld-debug进程，这可以让我在调试期间生成一个踪迹文件。

接下来要做的是启动MySQL客户端程序。注意，这里也需要通过--connect-timeout参数把通信等待时间设置为一个比较大的数值。下面是用来从一个命令窗口启动MySQL客户端程序的命令：

```
mysql -uroot -p --connect-timeout=600
```

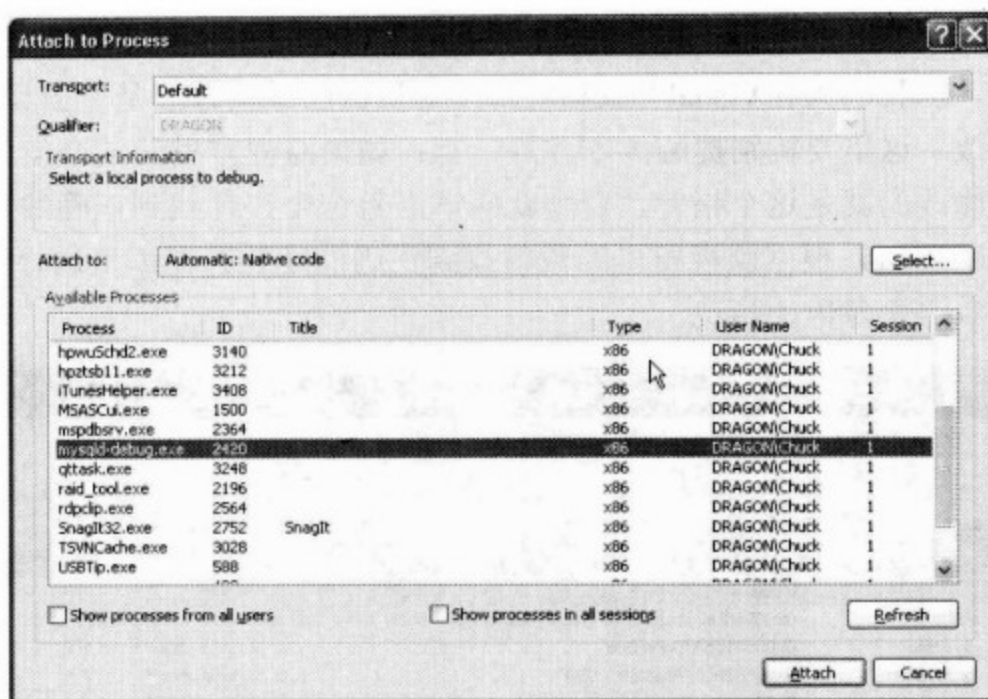



图5-9 在Visual Studio .NET里连接一个进程

启动MySQL客户端程序后，我用了它发出了show authors; 命令，Visual Studio将在它执行到我预先设置的断点时暂停它的执行。现在，可以使用“逐过程”（F10功能键）和“逐语句”（F11功能键）命令来单步执行各条语句。我在进入那个负责发送数据的循环后停下来查看了一下authors结构。图5-10给出了Visual Studio调试器在进入循环后的状态。

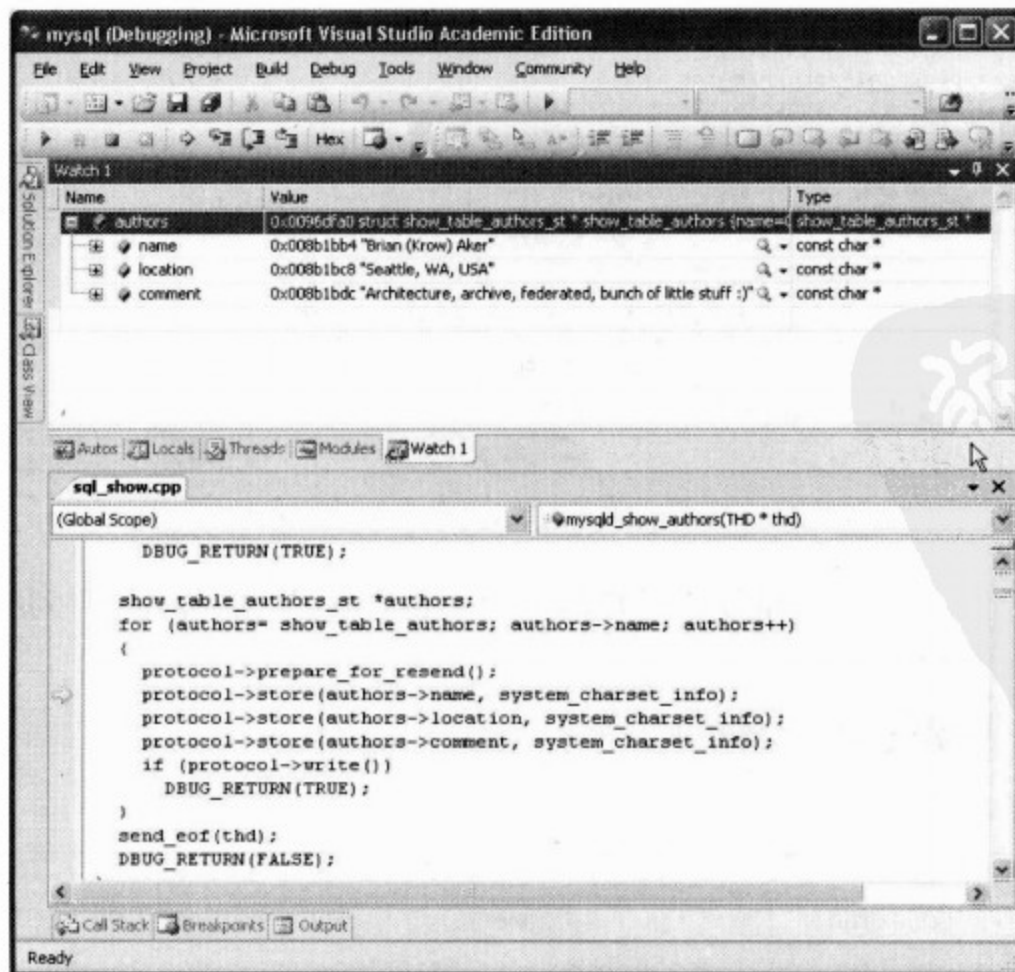


图5-10 在Visual Studio .NET里显示变量的值

类似于ddd调试器，在Visual Studio调试器里也可以修改变量的值，只是在细节上稍微有点儿麻烦。也许还有其他的办法，但我认为在Visual Studio里修改变量值的最佳办法是在监视窗口（名为Watch ...的子窗口）里进行修改。这里要特别提醒大家注意一点：如果监视窗里的某个值是一个指向内存地址的指针，除非你想要修改的就是这个指针，否则就应该沿着这个指针找到正确的地址再做修改。具体做法是：打开内存调试窗口，根据监视窗里的数值找到正确的地址并在那里进行修改。图5-11演示了通过内存调试窗口去修改变量值的情况。

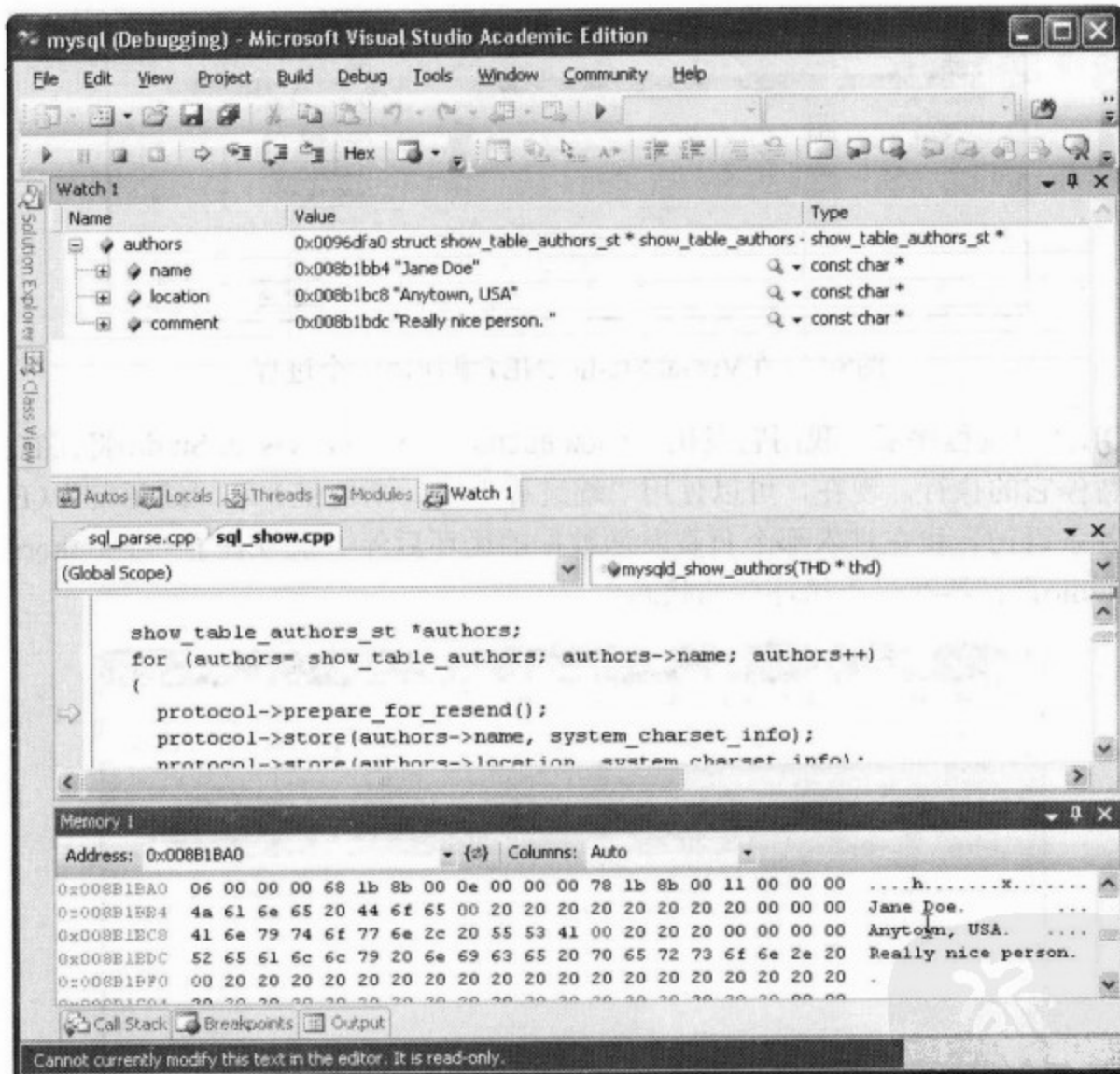


图5-11 在Visual Studio .NET里修改内存里的值

修改好内存里的值之后，可以继续执行并在MySQL客户端程序的窗口里看到结果。代码清单5-17给出了一份示例输出（有删节）。

代码清单5-17 本次调试会话的结果输出

```
mysql> show authors;
```

Name	Location	Comment
Jane Doe	Anytown, USA	Really nice person.

```
...
+-----+-----+-----+-----+
74 rows in set (1 min 55.64 sec)

mysql>
```

调试任务完成后，先在Visual Studio里用菜单命令Debug | Detach All解除Visual Studio调试器与MySQL服务器进程的关联关系，然后在一个命令窗口里发出shutdown命令关闭服务器^①。

```
mysqladmin -uroot -p shutdown
```

好了，既然已经知道如何在Windows平台上用Visual Studio去调试MySQL系统了，我建议读者再认真研究一遍这个例子并用它在自己的Windows开发机器上做个练习。

5.4 小结

本章解释了什么是调试，介绍了一些基本的调试技术，并给出一些关于如何使用这些技术的例子。这些技术包括内嵌调试语句、出错处理和外部调试器。

本章还讲述了如何使用MySQL AB公司提供的内嵌调试语句DEBUG工具来创建一个用于记录系统执行轨迹的踪迹文件，如何写出能帮你调试程序的数据，如何记录出错消息和警告消息等。文中还讲解了如何在Linux和Windows环境里使用gdb、ddd和Visual Studio .NET来调试MySQL系统。

要想成为一名优秀的开发人员，必须具备良好的调试技能。希望本章内容能够帮助大家进一步提高自己的调试技能。

下一章将对MySQL系统在系统集成商(或个人)当中最流行的用途之一进行讨论：嵌入式MySQL。简单地说，就是把MySQL系统变成另一个系统的一部分。不难想象，这个过程中可能需要某种严格的调试，以便找出在哪个嵌入层次上出现了什么问题。

① 原书中的操作顺序——先关闭服务器、后退出调试器——是公认的不良习惯。一般原则是：最后打开的程序或文件应该最先关闭。此处已经进行了更正。——译者注

MySQL服务器以体积小、性能高闻名于世，但你知道它还可以用作大型软件里的嵌入式数据库系统吗？本章将对嵌入式应用软件的概念，以及如何使用MySQL C API来创建自己的嵌入式MySQL系统进行解释和讨论。本章将介绍用来编译嵌入式MySQL服务器和为Linux、Windows平台编写这类应用软件的技巧。

6.1 构建嵌入式应用

有许多应用软件使用小型数据库系统作为其内部的数据存储机制。如果你是一位Windows用户，那你起码应该见过或用过一个使用Microsoft Access数据库引擎的应用软件——就算那些应用软件没在广告里说它们使用着Access，你也可以从它们的安装目录里看出端倪。

有些嵌入式应用软件使用的是其宿主计算机里现有的数据库系统（如Access），有些应用软件使用的是专门安装的比较大型的数据库系统，还有少数应用软件则是把数据库系统编译到自己的内部。

6.1.1 什么是嵌入式系统

所谓嵌入式系统（embedded system）是一个包含在另一个系统中的系统。简单地说，嵌入式系统是其宿主系统的一个子系统或从属系统。嵌入式系统的用途是为其宿主系统提供一些它所需要的功能，如某种通信机制、某种数据存储和检索机制，甚至是一个图形化的用户操作界面。

传统意义上的嵌入式系统都是些专用的硬件或电子设备。比如说，银行的自动柜员机就是一种包含着专用硬件的嵌入式系统。如今，嵌入式系统不仅包括专用的硬件设备，还包括专用的软件系统。与嵌入式硬件设备往往难以甚至不可能修改的情况不同，嵌入式软件一般都可以根据特定的环境进行定制。嵌入式硬件设备和软件系统的共同特点主要有两个：一是自我包含；二是都能为其宿主系统提供一些服务。

嵌入式软件系统与我们日常使用的各种应用软件是不一样的。为了让它们能在另一个软件系统里高效率地运转，往往需要对它们进行某种修改和定制。嵌入式MySQL服务器与它的独立型版本最大的区别在于，前者从一开始就被设计成需要通过编程来访问和操作。换句话说，对嵌入式MySQL服务器的访问，需要通过某种程序设计语言而不是SQL查询命令来进行。嵌入式MySQL系统的对外函数会把SQL查询命令当作参数传递给服务器去执行。

这意味着嵌入式MySQL服务器只能通过另一个应用程序去访问。不过，就像你将在接下来几个小节里看到的那样，嵌入式软件可存在于大量的应用中，根据其集成程度，这些应用从只能通过编程访

间的闭包，到被宿主应用“隐藏”的全功能系统，包含范围非常之广。

6.1.2 嵌入式系统的种类

嵌入式系统可以分为许多种类，但因为每一种都有自己独特的用途，所以很难对它们进行准确的分类。一般来说，一个嵌入式系统通常属于以下几大类别中的某一个（或某几个）：

- ❑ 实时系统（Real-time），这类嵌入式系统多见于需要宿主系统的某个部分在给定的阈值内作出快速响应的装置。对这类嵌入式系统最常见的要求是速度越快越好，处理每条命令所花费的时间必须压缩到最短才能实现整个系统的目标。相比内部处理速度，这类系统通常更需要处理外部事件的能力。实时系统的典型例子包括路由器和电话交换机。
- ❑ 被动响应系统（Reactive），这类嵌入式系统只能对外部事件做出响应。它们所响应的事件往往是重复或循环形式的，但也有一些是对用户输入做出响应（交互式系统都属于被动响应系统）。人们对被动响应系统的要求通常是必须能够长时间地连续运转，速度问题往往是第二位的。当然，那也要看它们所响应的事件重复或循环发生的频率有多高。被动响应系统的典型例子包括那些用于网页或必须在某些事件发生或超过阈值时通知相关人员的安监系统。
- ❑ 过程控制系统（process control），这类嵌入式系统的主要用途是控制其他系统的运转。这类嵌入式系统多见于需要对某种硬件设备（比如机械手或流水线设备）进行监控的场所。这些系统通常只会重复一组动作，一旦定型就不再改变，对外界发生的事件或者条件或变量的阈值不闻不问。过程控制系统的典型例子包括在汽车自动装配线上安装某种特定零部件的机械手或机器人。
- ❑ 高风险系统（critical），这类嵌入式系统多见于安全保卫部门、医疗单位或航空领域。人们对这类嵌入式系统的要求通常是不允许出现失误（最好是永远也不失误）。这类系统大都是刚才介绍的那几种嵌入式系统的某种变体。高风险系统的典型例子包括心脏起搏器和人工呼吸机等医疗设备。

6.1.3 嵌入式数据库系统

嵌入式数据库系统（embedded database system）是一种为其宿主软件或环境提供数据的系统。它们的宿主系统通常都对数据的提供速度有比较高的要求，数据库在接到指令后必须立刻把信息提供出来。嵌入式数据库系统的好坏不仅对其宿主应用软件关系重大，对整个全局的意义往往也非常重要。因此，嵌入式数据库系统通常还必须满足其用户对速度的要求。这些要求意味着嵌入式数据库系统应该被划分到被动响应系统的类别里。

不管是个人还是企业使用的应用软件，可以说没有不产生数据、不使用数据、不存储数据的。大部分应用软件都有着良好的数据结构，而存储在这些数据结构里的数据对它们的用户来说往往有着难以估量的价值。不过，很少会有用户关心自己的数据是如何存储和检索的（有相当多的人只知道这些操作已经自动化了），他们只关心自己的数据能不能在他们需要的时候立刻“出现”。像这样的应用软件通常都有一个子系统在扮演着“数据库服务器”的角色，这个子系统可以是一些数据存取函数、一些外部数据文件或是一个货真价实的数据库服务器。

使用文件来保存数据的嵌入式系统需要解决许多问题。别的先不说，它们至少应该保证其宿主系统以外的其他程序不能访问那些数据。但要切实做到这一点并不容易，你往往需要从零开始构建它的

访问权限控制机制，或在这个系统里再增加一个层次。虽然现代文件系统访问数据的性能和速度早已不是什么问题，但它们的灵活性比不上数据库系统。在数据存储方面，数据库系统要更灵活；但访问数据的速度较慢。

不同的人会因为不同的理由而希望自己的数据安全无忧，但基本的要求只有一个：就是如何在把数据泄露给别人的前提下，以最高的效率来存储和检索它们。对绝大多数情况而言，这个问题的答案非常简单：使用一个数据库系统。比如说，Adobe Production软件里有一个名为Adobe Bridge的组件，后者管理着前者需要用到的文件、项目、图片以及其他许多东西的大量数据。为了方便用户检索这些数据，有关文件需要以一种有条理的方式组织起来。Adobe使用了一个嵌入式数据库（MySQL）来管理由Adobe Bridge负责收集和保存的各种文件的元数据。在这种情形下，这个软件使用了一个数据库系统来完成存储和检索那些元数据的繁重工作。

从确保数据安全的角度看，使用一个外部的数据库系统往往不是最佳的解决方案，因为它很难把数据完完全全地保护（或隐藏）起来。嵌入式数据库系统填补了这个空白，在向其宿主应用软件提供全套数据库功能的同时，它把数据和数据的存储/检索机制在外部世界的眼前隐藏了起来。

6.2 嵌入 MySQL

MySQL AB公司早在开发MySQL之初就预计到它的许多顾客都是系统集成商，而这些顾客需要的是一种健壮、高效并可以通过编程进行访问的数据库系统。为满足这类顾客的需要，MySQL AB公司不仅推出了一个嵌入式库，还配套推出了一个名为libmysql的全功能的客户端程序库。该库允许用户创建自己的MySQL客户端程序。比如说，你可以创建一个MySQL命令行客户端程序版本。如果你想知道一个典型的MySQL客户端程序是如何使用这个库的，打开mysql项目源代码文件就可以看到。

MySQL服务器的可执行文件的名称是mysqld，嵌入式MySQL服务器开发库的名称是libmysqld。也有人把这个库称为“嵌入式服务器”或“C API”，本章只用“嵌入式程序库（libmysqld）”来称呼它。客户端程序和嵌入式服务器程序库在访问和连接上有很多相似之处。

嵌入式程序库提供了许多通过API（Application Programming Interface，应用程序接口）来访问MySQL数据库系统的函数。通过这个API，宿主系统可以（通过编程）充分利用MySQL服务器的强大功能。这些功能包括（但不限于）以下这些：

- ☐ 创建和连接一个服务器实例；
- ☐ 与服务器断开连接；
- ☐ 安全地关闭服务器；
- ☐ 调整服务器的启动选项；
- ☐ 处理各种错误；
- ☐ 生成 DBUG 踪迹文件；
- ☐ 发出查询命令和检索查询结果；
- ☐ 管理数据；
- ☐ 使用 MySQL 服务器的（几乎）全部功能。

最后一点是独立型服务器和嵌入式服务器最大的区别之一。嵌入式服务器没有使用完整的身份验证机制，而且在默认的情况下还是禁用的。这是嵌入式MySQL服务器相对不够安全的原因之一（详见

6.2.3节的讨论)。读者可以通过使用配置选项`--with-embedded-privilege-control`并重新编译嵌入式MySQL服务器的办法来启用身份验证机制。除了这一点,嵌入式MySQL服务器与独立型MySQL服务器在行为和功能方面几乎一模一样。

嵌入式服务器与独立型服务器使用的数据存取方法是一样的,所以嵌入式服务器可以直接使用用户使用独立型服务器创建的所有数据库和表。也就是说,你可以先用独立型服务器来创建表并测试它们,然后再把它们复制到嵌入式服务器下使用。请注意,让独立型服务器和嵌入式服务器共享同一个数据子目录并不是不可以,但因为这有可能导致数据丢失或难以预料的行为,所以MySQL AB公司非常反对这么做。(任何两个MySQL服务器实例都不应该共享同一个数据目录。)

这是否意味着你可以在同一台机器上将一个独立型服务器作为嵌入式服务器运行呢?不但可以,而且还可以运行多个。事实上,只要不让它们共享同一个数据目录,你可以在同一台机器上运行多个嵌入式服务器的实例。再强调一次,不要让两个嵌入式服务器的实例共享数据,应该把它们各自管理的数据隔离开。这一招我已经在系统上试过很多次,每次都很成功。我的机器里正运行着一个基于MySQL 5.0.22 GA版的嵌入式应用程序和一个5.19-β版的独立型MySQL服务器,用不着关闭独立型服务器就可以使用嵌入式服务器。够酷的吧?

注解 MySQL AB公司在2006年度的MySQL用户大会上承认,5.1版的嵌入式服务器有时不能正常地工作。不过,5.0版源代码里的嵌入式服务器代码不存在这个问题。本章里的所有例子都取材于5.0.22 GA版的服务器源代码。读者看到这本书的时候,MySQL AB公司应该已经把5.1版里的问题解决了。如果真是那样,本章中的例子应该与最新以及未来的5.1版源代码相兼容。

6.2.1 嵌入MySQL的方法

嵌入式应用软件有许多种类型。按照它们在宿主系统里的嵌入深度,嵌入式数据库应用软件通常可以分为三类:第一类是数据库系统只是部分地隐藏在另一种操作界面的后面(服务器级嵌入);第二类是数据库系统是由专用硬件和软件构成的某种专用设备的一部分,该设备与网络相隔离(平台级嵌入);第三类是数据库系统完全包含在其宿主系统的内部(深度嵌入)。下面将以MySQL系统为例,对这三种情况进行描述。

1. 服务器级嵌入

服务器级嵌入(server embedding)是由独立型MySQL服务器构建的系统,因为不想把数据库开放给整个系统或网络上的每一个人,宿主系统通过禁用外部(网络)访问功能的办法把MySQL服务器隐藏了起来。换句话说,这种级别的嵌入式MySQL服务器就是一个独立型MySQL服务器,只是它的网络访问功能(TCP/IP)被禁用了。

这种嵌入式MySQL系统的优点是,允许人们通过本地安装(并适当配置)的客户端应用程序去访问和管理MySQL服务器。这意味着我们不必使用外部应用程序去加载数据,系统集成商、系统管理员和程序员可以使用各种现成的系统管理工具和开发工具去管理和开发这种嵌入式MySQL服务器。

LeapFrog公司推出的LeapTrack软件(www.leapfrogschoolhouse.com/do/findsolution?detailPage=overview&name=ReadingPro)是服务器级嵌入式MySQL系统的一个典型例子。根据MySQL AB公司的

说法, LeapFrog公司之所以选用MySQL作为其跨平台支持, 是因为这可以让它们的产品在多种平台上使用而无需修改其核心的数据库功能。在选择MySQL之前, LeapFrog公司曾尝试过许多种专利数据库解决方案来解决其产品的跨平台运行问题, 但效果都不甚理想。

2. 平台级嵌入

平台级嵌入 (platform embedding) 要比服务器级嵌入有更多的限制。这种嵌入式系统使用的仍是一个独立型MySQL服务器, 但这个MySQL服务器已经与外界彻底隔离了, 访问这个MySQL服务器的唯一办法是通过其宿主系统提供的客户端接口。此时, 外部应用程序仍可以与MySQL服务器直接进行通信, 只是必须通过客户端程序提供的API来进行而已; 客户端程序提供的API相当于通往MySQL服务器的一扇大门。

在平台级嵌入情况下, 将由宿主系统负责提供和管理数据库子系统的访问和管理机制。还好, 许多有着特殊用途的系统管理和维护工具 (比如那些用来修复InnoDB表的工具) 仍可以照常使用和正常工作, 只有从客户端程序去访问数据库的通道受到了限制 (需要通过API来进行)。

Sandstorm公司推出的NetIntercept解决方案 (www.sandstorm.net/products/netintercept) 是平台级嵌入式MySQL系统的一个典型例子。NetIntercept产品是一种全功能的机柜安装式建网设备, 主要用来搭建高流量的网络服务器。顾客购买的NetIntercept系统是一个2U或4U的板卡式计算机系统, 只需插入机柜就可以接入网络。使用MySQL作为一个嵌入式平台使得Sandstorm公司的顾客可以享受到MySQL系统的好处, 但无需另外购买、安装和配置一套MySQL系统。Sandstorm公司把MySQL数据库封装 (或隐藏) 在它们自己的系统内部, 最终用户可能永远也不会知道MySQL是NetIntercept产品的一个子组件。

注解 2U中的“U”指的是19英寸机柜中的一个标准插槽。2U占用两个插槽位置, 4U占用四个插槽位置。

3. 深度嵌入

深度嵌入 (deep embedding) 比平台级嵌入还要严格。这种嵌入式系统把MySQL系统用作一个集成组件, 被嵌入的MySQL系统不仅无法从网络直接访问, 连各种常用的MySQL客户端应用程序也不能直接访问。这种嵌入式系统是用MySQL AB公司专门为此推出的嵌入式MySQL服务器开发库 (libmysqld) 搭建的。绝大多数嵌入式MySQL系统都属于这一类。

因为这种嵌入式系统仍通过MySQL来实现其数据访问机制, 所以它的数据库功能与独立型MySQL服务器基本相同, 只是有少数功能受到了限制 (稍后再讨论这个问题)。程序员可以使用许多种程序设计语言为许多种平台开发一个深度嵌入的MySQL系统 (请参见前面的讨论)。libmysqld库向程序员提供了其他任何关系数据库系统都无法提供的代码级解决方案。

使用一个深度嵌入式MySQL系统的最大好处是它可以提供一个与外界几乎完全隔离的MySQL系统, 让被嵌入的MySQL系统只向其宿主系统提供服务。

Adobe公司推出的Adobe Bridge (www.adobe.com/creativesuite/bridge.html) 是深度嵌入式MySQL系统的一个典型例子。Adobe Bridge是Adobe Creative Suite软件的一个组成部分, 其用途是对Creative Suite所支持的各种数据进行管理。如果你是一位Creative Suite的最终用户, 在看到本书之前,

你知道你使用的其实是一个专用的MySQL系统吗？^① 深度嵌入系统大都是些桌面软件，需要用户安装在他们的本地机器里运行。

6.2.2 资源要求

运行嵌入式服务器所需要的资源取决于它的嵌入深度。如果你使用的是服务器级嵌入或平台级嵌入，所需要的资源将与独立型服务器一样。深度嵌入的MySQL系统就不同了。深度嵌入的MySQL系统一般需要2MB左右的内存——它的宿主系统所需要的内存不包括在内。经过编译的嵌入式服务器会让可执行文件的内存占用量稍微加大一些，但一般不至于成为负担或无法管理。

磁盘空间是大概是最难以准确测算的资源了，因为它取决于嵌入式系统将要使用的数据量有多大。磁盘空间和时间也是人们在开发高数据流量系统和/或高数据修改量系统时最关心的问题之一。大量的数据修改操作对响应时间的影响往往要比对磁盘空间的影响来得大。此时，为了让系统管理员能及时地对数据库进行管理和维护，嵌入式系统往往需要为系统管理员提供专用的服务器访问通道或专用的接口。这种专用通道在服务器级或平台级嵌入系统里相对比较容易实现，在深度嵌入系统里就比较困难了。

6.2.3 安全问题

嵌入式系统的安全性取决于它的嵌入深度。服务器级嵌入系统的安全问题最具挑战性——因为嵌在宿主系统里的MySQL服务器，仍允许人们通过安装在本地主机里的各种工具进行访问，所以在这类嵌入式系统上要想彻底杜绝安全隐患是非常困难的。

平台级嵌入系统的安全问题就容易解决得多了，虽然被嵌入的是一个独立型MySQL系统，但人们只能通过其宿主系统才能访问它。除非嵌入式应用软件的开发者心怀叵测，否则肯定会采取一些措施来控制用户（尤其是高权限用户）的访问权限。

深度嵌入系统上最大的安全问题是如何保护好数据。这类系统里的MySQL子系统往往没有任何预设的密码。这是因为既然用户“只能”通过宿主系统提供的接口去访问数据库，为什么还要多此一举呢？可是，事情并没有这么简单。有不少深度嵌入系统都把数据存放在用户可以直接访问的子目录里。实际上，如果不允许用户访问那个子目录，他们又该如何读写他们的数据呢？

问题就出在这里：数据文件没有受到保护意味着某人可以把它们复制到另一台安装着MySQL软件的机器上为所欲为。其实这类问题并不仅限于嵌入式系统，独立型服务器也存在着类似的隐患。也许你从未想到过这一点，但如果你们公司在使用开源软件方面没有什么严格的规章制度的话，这种事情就难免会发生。想想看，如果你向公司负责信息安全的那位老兄指出这一点，他的表情会变成什么样？当然，这种事你应该说得尽量婉转点儿。总而言之，在设计和开发一个嵌入式系统的时候，一定要把信息安全问题考虑进去，一定要增加必要的安全措施，以保护被嵌入的MySQL服务器和数据库里的数据。

6.2.4 嵌入MySQL的优点

MySQL嵌入式API使得开发人员可以在另一个应用程序的内部使用一个全功能的MySQL服务器。最主要的优点包括：提高数据访问速度（因为MySQL服务器是其宿主系统的一部分，或者与其宿主系

^① 在写作本书时好像还是。

统运行在同一台机器里)，有内建的数据库管理工具，有一个非常灵活的存储和检索机制。这些优点可以让程序员在充分利用MySQL的优势开发软件产品的同时，把它的具体实现在用户眼前隐藏起来。这意味着程序员可以借助于MySQL的功能而提高自己产品的能力。

6.2.5 嵌入MySQL的局限性

使用嵌入式MySQL服务器也有一定局限性。还好，这份清单并不长，而且绝大多数局限性都有合理的原因，对系统集成商来说不是什么问题。表6-1列出了嵌入式MySQL系统的一些已知局限性，并对它们分别做了简要的描述。

表6-1 嵌入式MySQL系统的局限性

局 限 性	说 明
安全	访问控制机制在默认情况下处于禁用状态。权限管理系统不起作用
镜像	没有镜像功能，也没有日志功能
外部访问	不允许与外界进行网络通信（除非你自行实现这部分功能）
安装	深度嵌入系统（如libmysqld）可能需要额外的库才能部署
数据	嵌入式服务器存储数据的做法与独立型服务器一样：每个数据库对应于一个文件夹；每个表对应于一组文件
版本	嵌入式服务器不能与MySQL 5.1.9b版配合工作，但在更高的版本里应该没问题
UDF	不允许使用用户定义函数（user-defined function，UDF）
调试/跟踪	在系统即将崩溃之前，可以生成内存映像文件，但不生成踪迹文件
连接性	你无法从网络协议连接到嵌入式服务器。不过，可以通过其宿主系统提供这种连接性
资源	如果是一个需要支持海量数据和/或许多并发连接的服务器级或平台级嵌入式系统，资源消耗量会很大

6.3 MySQL C API

在第一次看到关于MySQL C API的文档（《MySQL参考手册》的第25章）时，许多人都会望而生畏。这也难怪，毕竟封装在MySQL C API里的是独立型MySQL服务器的全部功能，想把事情面面俱到地解释清楚当然需要些篇幅。还好，我们随时都可以到MySQL AB公司的网站上查阅MySQL的在线文档（<http://dev.mysql.com/doc>）。

注解 MySQL的在线文档通常是关于其最新版本的。如果你为了阅读方便而下载过这份文档，应该定期检查一下下载的副本是否已经过时。每当我遇到难题的时候，都会去查看一下MySQL在线文档，而且每次都会有所收获。

颇有讽刺意味的是，MySQL C API最令人望而生畏的地方应该只是那份文档本身。简单地说，它有点儿过于简练，经常需要反复阅读好几遍才能把某个概念弄明白。在接下来的几个小节里，将简单讲解一下这个C API的概况，再通过几个具体的例子帮助大家尽快开始自己的嵌入式应用程序项目。

6.3.1 预备知识

对于那些希望学习如何开发嵌入式应用程序的程序员朋友，我的第一个建议是阅读相关的文档。

只学习书本上的知识是不够的，在开始使用一个API之前，应该先通读几遍它的随机文档，就算一时半会儿用不上它们也没关系。我在查阅MySQL文档的时候，经常遇到这样的情况：有些信息第一次看见时觉得没什么用，后来却成为一次成功的编译和几小时徒劳无功的调试之间的分水岭。

我的第二个建议是应该经常到MySQL AB公司的论坛网站（在<http://forums.mysql.com>上有一个专门讨论嵌入式系统的论坛）和邮件表网站（<http://lists.mysql.com>）上去看看。用不着阅读所有的帖子，但如果你遇到什么问题的话，答案往往能在那些帖子中找到。MySQL的博客网站（www.planetmysql.org）也值得去看看，那里有来自各行各业的作者发表的探讨嵌入式系统和许多其他问题的文章。那些文章有的写得确实精彩，我常常一看就是好几个小时。有不少MySQL专家就是通过在这些网站上阅读和撰写文章，才成为大家公认的高手的。信息就是力量。

MySQL的在线文档以及各种各样的MySQL邮件列表和博客，是了解MySQL最新发展动向的最佳场所。你们应该知道的最重要的东西，都包含在接下来的几个小节里。在介绍了MySQL C API里的几个最重要的函数之后，还准备了一个简单的例子来演示如何编写嵌入式应用程序。在那之后，将展示一个更为复杂的嵌入式应用程序，它使用了一个抽象的数据类，并且是在.NET环境里编写的。

学习如何创建嵌入式应用程序的最佳办法是亲自动手编写一个。请打开你最喜欢的源代码编辑器，并跟着下面给出的几个例子实践一下。我将先按照它们在程序代码里的调用顺序来介绍几个最常用的函数，然后在下一小节向大家介绍如何构建程序库，如何编写你们的第一个嵌入式服务器应用程序。

6.3.2 最常用的函数

我大致统计了一下，MySQL C API支持的函数至少有65个。其中有一部分函数已经过时了，MySQL AB公司也在文档里做了相应的说明。总的来说，这个API里只有不多的几个函数是常用的。

这个API里的绝大多数函数都与建立/断开连接和管理/维护服务器有关。有些函数专门用来收集关于服务器和数据的信息，还有些函数专门用来执行查询命令或进行其他的数据处理，还有几个函数是专门用来检索出错信息的。

表6-2列出了这个API里最常用的函数并对它们做了简单的描述。这些函数在表6-2里的排列顺序，与它们在一个简单的嵌入式服务器程序里的调用顺序基本一致。

注解 在学习完本章的例子之后，我希望大家能抽时间读一下《MySQL参考手册》里介绍C API的章节。如果你的数据库有特殊要求，说不定能在那些章节里发现一些你正愁没地方找的函数。

表6-2 MySQL C API (libmysqld) 里最常用的函数

函 数	说 明
<code>mysql_server_init()</code>	对嵌入式服务器库进行初始化
<code>mysql_init()</code>	启动服务器
<code>mysql_options()</code>	用来改变或设置服务器选项
<code>mysql_debug()</code>	启用调试踪迹文件 (DEBUG)
<code>mysql_real_connect()</code>	与嵌入式服务器建立连接
<code>mysql_query()</code>	发出一个查询命令 (SQL语句)，其格式为一个以空字符 (null) 结尾的字符串

(续)

函 数	说 明
mysql_store_results()	检索最后一个查询命令的结果
mysql_fetch_row()	返回结果集里的某一行
mysql_num_fields()	返回结果集里的列(字段)个数
mysql_num_rows()	返回结果集里的行(记录)个数
mysql_error()	返回一条出错消息(字符串), 它描述的是最近发生的那个错误
mysql_errno()	返回一个出错代码, 它对应于最近发生的那个错误
mysql_free_result()	释放分配给指定结果集的内存。注意: 你们应该及时释放已经不再需要的结果集; 不要害怕调用这个函数的次数太多——即使你已经释放过一个结果集, 再次释放它也不会导致错误
mysql_close()	关闭与服务器的连接
mysql_server_end()	对嵌入式服务器库进行关机处理, 然后关闭服务器

对这些函数的详细描述(返回值、语法等)见《MySQL参考手册》。

6.3.3 创建嵌入式服务器

嵌入式服务器在初始化函数调用期间被创建为一个实例, MySQL C API里的绝大多数函数都需要一个指向这个服务器实例的指针作为必要的参数。在创建嵌入式应用程序的时候, 你需要创建指向MySQL对象的指针。还需要为结果集和结果集里的行(也叫做“记录”)分别创建实例。服务器和这几种数据结构的定义都是在MySQL头文件里完成的。你们需要用到的两个头文件(绝大多数应用程序也只需要用到这两个文件)是。

```
#include <my_global.h>
#include <mysql.h>
```

创建指向嵌入式服务器、结果集和记录结构的指针的工作可以用如下命令来完成。

```
MYSQL *mysql;           // the embedded server class
MYSQL_RES *results;     // stores results from queries
MYSQL_ROW record;       // a single row in a result set
```

这些语句使得你能够在后面的代码里访问嵌入式服务器(MYSQL)、一个结果结构(MYSQL_RES)和一条记录(MYSQL_ROW)。你可以使用全局变量来定义这些指针; 但如果你不喜欢使用全局变量, 也完全可以把它们定义成局部变量。结果集和记录可以随时创建和释放, 唯一需要注意的地方是, 必须保证MYSQL指针变量在你的应用程序里是同一个实例。

我们还没有完成初始化工作。还需要创建一些字符串供建立连接时使用。创建这些字符串的办法有许多种, 但最常见的办法是创建一个字符串数组。你至少需要为my.cnf文件(在Windows平台上是my.ini文件)的存放位置和数据存放位置分别创建一个字符串。下面是一组典型的初始化字符串。

```
static char *server_options[] = {"mysql_test",
    "--defaults-file=c:\\mysql_embedded\\my.ini",
    "--datadir=c:\\mysql_embedded\\data" };
```

本章中的例子给出的是适用于Windows编译的服务器选项。如果你使用的是Linux, 将需要使用相应的路径, 并把my.ini改为my.cnf。在这个例子里, 使用了标签mysql_test (mysql_server_init())函

数将忽略它), my.cnf (my.ini) 文件的存放位置是普通的安装目录, 数据子目录的存放位置是普通的MySQL安装。如果你想在同一台机器上同时运行一个独立型服务器和一个嵌入式服务器, 应该为每个服务器分别指定不同的数据存放位置。为简明起见, 还应该为每个服务器分别使用不同的配置文件。

为了把发生错误的概率降到最低, 我还使用了一个整数来给出字符串数组里的元素个数(过一会儿还要讨论这个问题)。这使我可以放心地编写边界检查代码, 而无须顾虑会弄错那些元素的个数。我可以那些元素在程序运行时发生变化, 并让边界检查代码根据这些变化进行必要的检查。

```
int num_elements=sizeof(server_options) / sizeof(char *);
```

最后一个初始化步骤是创建另外一个字符串数组来确定服务器组从我的配置文件(my.cnf)里读取所有额外的服务器选项, 这就定义了服务器启动时将被读入的节。

```
static char *server_groups[] = {"libmysqld_server", "libmysqld_client" };
```

6.3.4 对服务器进行初始化

只有在初始化或启动嵌入式服务器之后, 才可以去连接它。这通常涉及两个初始化调用和一组用来设置其他必要选项的调用。为了启动一个嵌入式服务器, 你需要调用的第一个初始化函数是mysql_server_init()。下面是这个函数的定义。

```
int mysql_server_init(int argc, char **argv, char **groups)
```

这个函数只需要在调用任何其他函数前调用一次。它的argc和argv参数与其他程序的这两个参数



提示 几乎所有的mysql_XXX()函数都在成功时返回0，在失败时返回非零值。只有那些返回指针的函数在成功时返回非零值，在失败时返回0 (NULL)。

6.3.5 设置选项

嵌入式服务器允许你在尝试连接服务器之前设置一些必要的连接选项。下面是你将用来设置连接选项的函数的定义。

```
int mysql_options(MYSQL *mysql, enum mysql_option, const char *arg)
```

第一个参数是嵌入式服务器对象的实例。第二个参数是你想设置的选项之一。最后一个参数是一个可选的字符串，它负责把第二个参数所指定的选项的设置值（如果有的话）传递进来。可供选用的连接选项以及它们的可取值可以列成一份长长的清单。表6-3列出了其中最为常用的选项和它们的值。连接选项的完整清单可以在《MySQL参考手册》里查到。

表6-3 连接选项的部分清单

选 项	值	说 明
MYSQL_OPT_USE_REMOTE_CONNECTION	N/A	连接到一个远程服务器
MYSQL_OPT_USE_EMBEDDED_CONNECTION	N/A	连接到一个嵌入式服务器
MYSQL_READ_DEFAULT_GROUP	组	让服务器从配置文件中的指定组里读取服务器配置选项
MYSQL_SET_CLIENT_IP	IP地址	为被配置成启用了身份验证机制的嵌入式服务器提供IP地址

下面是这个函数的调用示例，第一个调用告诉服务器从配置文件的[libmysqld_client]节读取配置选项，第二个调用告诉服务器使用一个嵌入式连接。

```
mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
```

如果执行成功，这个函数的返回值将是0；如果某个选项无效或某个选项的设置值无效，将返回一个非零值。

6.3.6 连接到服务器

现在，服务器已经过初始化并设置好了所有的选项，可以连接服务器了。用来完成这项任务的函数是mysql_real_connect()。它有许多参数，可以通过这些参数对连接的细节进行调控。下面是这个函数的定义情况。

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const
char *passwd, const char *db, unsigned int port, const char *unix_socket,
unsigned long client_flag)
```

这个函数必须无任何错误地完成。如果它失败了（事实上，如果此前的任何一个函数失败了），你将无法使用服务器，而只能重新尝试连接服务器或体面地结束这次操作。

这个函数的参数包括：你刚才创建的MYSQL实例、一个主机名字符串（IP地址或URL地址均可）、一个用户名、进入服务器后将使用的第一个数据库的名字、你打算使用的端口号、你打算使用的Unix

套接字、最后是一个用来激活各种客户端行为的标志。对客户端行为激活标志的详细介绍见《MySQL 参考手册》。如果传递给某个参数的值是NULL，这个函数将使用该参数的默认设置值。下面是这个函数的一个调用示例，除了进入服务器后将使用的第一个数据库的名字，其他参数都将使用默认的设置值。

```
mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema", 0, NULL, 0);
```

如果执行成功，这个函数将返回一个连接句柄；如果执行失败则返回NULL。绝大多数应用程序不需要捕获那个连接句柄，但应该检查这个函数的返回值是不是NULL。请注意，在上面这个调用示例里，我没有使用任何身份验证参数，这是因为身份验证机制在默认情况下是禁用的。如果当初在编译这个嵌入式服务器时打开了身份验证开关，现在就需要提供这些参数了。最后，第四个参数是你想连接的默认数据库的名字；这个数据库必须已经存在，否则你将遇到错误。

此时，调用嵌入式服务器所需要的变量、对服务器进行初始化、设置连接选项、连接嵌入式服务器的代码片段都已经有了。以下是之前的代码示例所给出的这些操作。

```
#include "my_global.h"
#include "mysql.h"

MYSQL *mysql;           //the embedded server class
MYSQL_RES *results;     //stores results from queries
MYSQL_ROW record;       //a single row in a result set

static char *server_options[] = {"mysql_test",
    "--defaults-file=c:\\mysql_embedded\\my.ini",
    "--datadir=c:\\mysql_embedded\\data" };
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmyswld_server", "libmysqld_client" };

int main(void)
{
    mysql_server_init(num_elements, server_options, server_groups);
    mysql = mysql_init(NULL);
    mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
    mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
    mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema",
        0, NULL, 0);

    ...

    return 0;
}
```

6.3.7 运行查询命令

经过上述准备工作之后，我们来到了数据库系统之所以是数据库系统的地方：处理即时查询。用来发出查询的函数是mysql_query()函数。下面是这个函数的定义。

```
int mysql_query(MYSQL *mysql, const char *query)
```

这个函数的参数是刚才创建的MySQL实例和一个包含着SQL语句的字符串（以空字符null结尾）。这条SQL语句可以是任何合法的查询命令，包括各种数据处理语句（SELECT、INSERT、UPDATE、DELETE、DROP等）。如果查询命令有需要返回的结果，结果数据将被绑定到一个指针变量，你需要通过这个指针变量调用mysql_store_result()和mysql_fetch_row()方法才能访问那些结果。如果你的查询命令没有需要返回的结果，结果集将被设置为NULL。

下面这个函数的一个调用示例，将把当前服务器里的数据库名单列出来。

```
mysql_query(mysql, "SHOW DATABASES;");
```

如果执行成功，这个函数将返回0；如果执行失败，将返回一个非零值。

6.3.8 检索查询结果

在发出查询命令之后，下一步是取回结果集，并把它的一个引用指针保存到专门为此而声明的结果指针变量里去。有了这个指针，你就可以取回下一个行（记录），并把它保存到记录结构（MYSQL_ROW，一个命名数组）里去了。用来完成这一过程的函数是mysql_store_result()和mysql_fetch_row()，下面是它们的定义。

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

mysql_store_result()函数接受一个MYSQL对象作为它的参数，并为最近一次执行的查询命令返回一个结果集实例。如果执行出错或最近一条查询命令没有返回任何结果，这个函数将返回NULL。在mysql_store_result()函数返回NULL的时候，你应该或者说必须调用mysql_errno()函数，检查一下是你的查询命令没有返回结果，还是mysql_store_result()函数在执行时出了错。如果是执行出了错，应该把mysql_error()或mysql_errno()函数的返回值与已知错误表进行对比来了解问题的原因。与mysql_store_result()函数有关的已知错误值包括CR_OUT_OF_MEMORY（没有内存来容纳结果集）、CR_SERVER_GONE_ERROR或CR_SERVER_LOST（与服务器的连接意外中断）、CR_UNKNOW_ERR（服务器处于无法预料的状态）。

注解 mysql_store_results()函数的用法有很多，这里描述的只是它最常见的用途。如果你想了解这个函数的其他用法，或是在使用这个函数时遇到了问题，请查阅《MySQL参考手册》里的有关内容。

mysql_fetch_row()函数接受一个结果集作为它唯一的参数。如果结果集里没有更多的行可供取回，这个函数将返回NULL。这是一个很方便的安排，因为它允许你在一个循环或遍历函数里使用这个功能。你应该安排一个mysql_errno()函数调用，在mysql_fetch_row()函数返回NULL之后检查一下是不是发生了错误。与mysql_fetch_row()函数有关的错误包括CR_SERVER_LOST（与服务器的连接意外中断）和CR_UNKNOW_ERROR（发生错误，但原因不明）。

下面这段示例代码将查询一个表，并把结果显示在控制台窗口里。

```
mysql_query(mysql, "SELECT ItemNum, Description FROM tblTest");
results = mysql_store_result(mysql);
while(record=mysql_fetch_row(results))
{
```



```
printf("%s\t%s\n", record[0], record[1]);
}
```

请注意，在发出查询命令之后，我调用mysql_store_result()函数取回了结果；然后把mysql_fetch_row()函数放在了一条循环语句的循环控制表达式里。因为mysql_fetch_row()函数在到达结果集的末尾时会返回NULL，所以那条循环语句将在那时退出循环。在结果集里还有行可供取回的时候，我使用了一个数组下标（从0开始计数）对行里的每一个列进行访问。

这个例子提供了一个让嵌入式服务器执行各种查询命令的基本框架。你可以把这个过程封装为一个更大的函数，并把它放到一个类或一个抽象函数集合里。我将在第二个示例嵌入式应用程序里演示它的具体做法。

6.3.9 清理

从查询命令返回并被放在结果集里的数据，需要系统为它们分配必要的资源。作为一名优秀的程序员，应该及时释放不再需要的内存以避免内存泄漏^①（memory leak）。MySQL AB公司提供了一个名为mysql_free_result()的函数来帮助释放那些资源。下面是这个函数的定义。

```
void mysql_free_result(MYSQL_RES *result)
```

这个函数允许调用任意次，即使某个结果集已经被释放了，再次调用这个函数去释放它也不会导致出错。这意味着可以毫无顾虑地使用这个函数。别以为这是笑话——我就见过free调用比new调用还多的程序。这一般不会成为什么问题，但如果free调用使用不当，过多地使用它们，就有可能导致一些不该被释放的东西也被释放掉了。与对待new操作一样，在使用free操作的时候也应该有针对性，而且必须多加小心。

下面是一个调用这个函数来释放一个结果集的例子。

```
mysql_free_result(results);
```

6.3.10 与服务器断开连接并关闭服务器

在用完嵌入式服务器之后，需要断开与它的连接并关闭它。这可以用mysql_close()和mysql_server_end()函数来完成；前一个函数负责断开连接，后一个函数负责对服务器进行关机处理和释放内存。下面是这两个函数的定义情况。

```
void mysql_close(MYSQL *mysql);
void mysql_server_end();
```

这两个函数的调用示例如下所示。请注意，这两个调用应该是你在即将关闭服务器之前（而这又往往发生在你即将退出整个应用程序之前）最后发出的两个调用。

```
mysql_close(mysql);
mysql_server_end();
```

6.3.11 汇总

现在，让我们把前几个小节里的讨论整理一下。代码清单6-1给出了一个完整的嵌入式服务器应用

① 内存当然不会真的泄漏，这里说的是某块内存已不再有程序在使用它，但因为未被释放而导致无法使用的现象。

程序，它将把给定数据子目录里的数据库名单列出来。稍后的小节里将介绍如何编译和运行这个例子。

注解 下面这个例子是为Windows平台编写的。在后面的讨论里还会给出一个适用于Linux平台的例子。

代码清单6-1 嵌入式服务器应用程序的例子

```
#include "my_global.h"
#include "mysql.h"

MYSQL *mysql;                //the embedded server class
MYSQL_RES *results;          //stores results from queries
MYSQL_ROW record;            //a single row in a result set

static char *server_options[] = {"mysql_test",
    "--defaults-file=c:\\mysql_embedded\\my.ini",
    "--datadir=c:\\mysql_embedded\\data" };
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmysqld_server", "libmysqld_client" };

int main(void)
{
    mysql_server_init(num_elements, server_options, server_groups);
    mysql = mysql_init(NULL);
    mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
    mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
    mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema",
        0, NULL, 0);
    mysql_query(mysql, "SHOW DATABASES;");                // issue query
    results = mysql_store_result(mysql);                  // get results
    printf("The following are the databases supported:\n");
    while(record=mysql_fetch_row(results))                // fetch row
    {
        printf("%s\n", record[0]);                        // process row
    }
    mysql_query(mysql, "CREATE DATABASE testdb1;");
    mysql_query(mysql, "SHOW DATABASES;");                // issue query
    results = mysql_store_result(mysql);                  // get results
    printf("The following are the databases supported:\n");
    while(record=mysql_fetch_row(results))                // fetch row
    {
        printf("%s\n", record[0]);                        // process row
    }
    mysql_free_result(results);
    mysql_query(mysql, "DROP DATABASE testdb1;");          // issue query
    mysql_close(mysql);
    mysql_server_end();
    return 0;
}
```

6.3.12 出错处理

你们也许会感到困惑：在前一章对出错处理进行了那么多的讨论，但在代码清单6-1里怎么看不到它们的影子呢？别着急，MySQL C API早已为我们准备好了必要的工具。MySQL AB公司用两个函数提供了出错处理机制。第一个是mysql_errno()函数，用来检索最近发生的错误的出错代码；第二个是mysql_error()函数，用来检索最近发生的错误的出错消息。下面是这两个函数的定义情况。

```
unsigned int mysql_errno(MYSQL *mysql)
const char *mysql_error(MYSQL *mysql)
```

这两个函数的输入参数都是一个MYSQL对象。因为这两个函数都是出错处理器，所以它们永远也不会执行失败。不过，如果你在没有发生错误的情况下调用了它们，mysql_errno()函数将返回0，mysql_error()函数将返回一个空字符串。

下面是这两个函数的用法示例。

```
if(somethinggoeshinkyhere)
{
    printf("There was an error! Error number : %d = %s\n",
        mysql_errno(&mysql), mysql_error(&mysql));
}
```

好了，到这里就全部结束了。我希望我的解释能够拨开MySQL参考手册里的迷雾。之所以编写这一节，是因为感觉同类书籍没有提供可以帮助大家了解如何编写和使用嵌入式服务器的例子——至少没有这么简明扼要的。

6

6.4 构建嵌入式 MySQL 应用程序

上一节介绍了用在一个嵌入式MySQL应用程序里的基本函数。本节将演示如何真正建立一个嵌入式MySQL应用程序。我将从如何编译这样一个应用程序开始，然后对如何构造各种嵌入式库调用的方法进行讨论。还准备了两个例子供读者在自己的系统上做这方面的练习和试验。

还将简要地涉及一些关于如何修改核心MySQL源代码的问题。是的，我知道这听起来有点儿吓人，但将一个步骤一个步骤地把所有的细节展示在读者眼前。还好，那不过是一个简单的修改，只需改动两个文件而已。

我建议大家认真阅读一下本节给出的源代码。它们虽然有点儿多，但我已对与本节讨论内容没有关系的部分做了最大限度的删节。我从MySQL的源代码里学到了许多有用的东西，学习方法就是反复地阅读它们。本节的目标就是让读者通过学习示例源代码，对如何建立自己的嵌入式MySQL应用程序有进一步的认识和体会。

6.4.1 编译 libmysqld 库

在使用libmysqld库开始工作之前，需要做的第一件事情是编译它。MySQL的二进制发行版本通常都不包括预编译的libmysqld库。libmysqld库被收录在绝大多数的源代码发行版本里，可以在源代码树根目录下的/libmysqld子目录里找到它。这个库在编译时一般都不带有调试信息。如果想用MySQL搞开发的话，应该为自己准备一个激活了调试功能的版本。

1. 在Linux环境下编译libmysqld库

在Linux环境下编译libmysqld库需要用configure脚本设置好配置，然后用make和make install命令进行编译和安装即可。将需要的编译参数是--with-debug和--with-embedded-server。完整的编译过程如下所示。你应该从MySQL源代码的根目录运行这些命令。编译过程需要花费一些时间，可以现在就开始编译，然后一边等它完成，一边继续阅读本书。根据机器的速度和你以前是否在调试模式下编译过这个系统，这大概需要几分钟到一个小时左右的时间。

注解 以下命令将编译服务器，并把它安装到默认的位置。这些操作需要root用户权限。

```
./configure --with-debug --with-embedded-server  
make  
make install
```

提示 编译选项的完整清单可以用./configure --help命令查看。

2. 在Windows环境下编译libmysqld库

在Windows环境下编译libmysqld库需要启动Visual Studio，并打开源根目录里的主解决方案文件（文件名是mysql.sln）。启用调试功能很简单：选择libmysqld项目并把编译选项设置为Embedded_debug win32即可。可以通过菜单命令Build ► Build libmysqld或者是通过编译整个解决方案的办法来编译这个库，所有相关的项目会根据需要被编译进来。编译过程需要花费一些时间，可以现在就开始编译，然后一边等它完成，一边继续阅读本书。根据机器的速度和读者以前是否在调试模式下编译过这个系统，这个过程也需要几分钟到一个小时左右的时间。

6.4.2 调试问题如何解决

你们也许正想知道在libmysqld库里进行调试与独立型服务器的情况是否一样。是的，完全一样，你可以使用同样的调试方法。调试一个正在运行着的嵌入式服务器颇有一些挑战性，但一般不会遇到需要调试到那种级别的情况——MySQL服务器是嵌在另一个系统里的，调试其宿主系统往往更简便易行。不过，为了帮助你调试整个应用程序，创建一个踪迹文件还是很有必要的。

上一章介绍了好几种调试技术。从功能和简便两方面考虑，DBUG工具可以说是最突出的。不过，虽然嵌入式服务器里的有关函数都包含着与独立型服务器同样的DBUG标记，但问题是libmysqld库没有把DBUG功能对外开放。

你可以创建自己的DBUG工具包实例并用它来生成踪迹文件，你可能通过使用嵌入式服务器来为大型应用程序执行这样的操作，大多数应用程序都小到所添加的工作无法起作用。在这种情况下，如果嵌入式库提供一个调试选项，那就太棒了。

可以通过配置文件或通过直接调用嵌入式库里的一个函数来启用DBUG工具包。当然，这样做的前提是嵌入式库在编译时激活了调试功能。

在运行时启用踪迹文件需要向嵌入式库发出一个调用。被调用的方法是mysql_debug()，它需要你为其传递一个包含着调试选项的字符串作为参数。下面的示例代码将在运行时启用踪迹文件，给出最常用的调试选项，并让该库把踪迹文件写到根目录下。这个方法应该在开始连接服务器之前调用。

```
mysql_debug("debug=d:t:i:0,\\mysqld_embedded.trace");
```


提示 应该为自己的嵌入式服务器的踪迹文件起一个与其他服务器的踪迹文件不同的名字。这可以帮助读者区分哪些踪迹属于嵌入式服务器，哪些踪迹属于正在使用的另一个服务器。

你还可以通过配置文件来调试功能。只要把上面这个例子中的字符串放到嵌入式服务器在启动时将读取的my.cnf (my.ini) 文件里就行了（后面还会对此做进一步讨论）。

如果你想在嵌入式应用程序里使用DEBUG工具包，但又不想把DEBUG工具包包括在自己的代码里，你该怎么办？是不是没有办法做到这一点呢？别着急，虽然嵌入式库没有对外开放DEBUG方法，但这并不等于说它不能这样做！下面几段演示了如何修改嵌入式服务器，以使它对外提供一个简单的DEBUG方法。这里将使用一个简单的例子，因为并不想这么早就讲解最深的内容。

需要做的第一件事情是为源代码制作一个备份。如果你当初下载的是一个*.tar或*.zip文件，可以省略这个步骤。如果你在修改了一些代码之后，陷入无法让服务器通过编译的僵局，那个备份可以让你减轻不少负担（和心理压力），尤其在删除了你的修改之后还无法让服务器通过编译的时候。

给libmysqld库增加一个新方法，对读者来说应该是小菜一碟。先在/include子目录里的mysql.h文件中加上新方法的定义。在接下来的例子里，将创建一个方法来对外提供DEBUG_PRINT函数。我把它命名为mysql_debug_print()。代码清单6-2是这个方法的函数定义。请注意，该函数接受一个字符指针作为参数，我将通过这个参数把预先定义的一个字符串传递到嵌入式应用程序里，这使我能够把一个字符串写入踪迹文件，它可以作为某种标志让我把宿主系统和嵌入式服务器的调试踪迹区分开。

6

代码清单6-2 修改mysql.h文件

```
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* Adds a method to permit embedded applications to call DEBUG_PRINT */
void STDCALL mysql_debug_print(const char *a);
/* END CAB MODIFICATION */
```

接下来是创建那个新函数。这需要编辑/libmysqld/libmysqld.c文件（在Windows环境下，对应的源代码文件名是/libmysqld/libmysqld.cc）并把新函数添加到这个文件中的源代码里去。把新函数添加到什么位置并不重要，只要它是在源代码的主体部分就可以。代码清单6-3是这个新方法的代码。请注意，该新方法只做了一件事：把字符串传递给DEBUG_PRINT函数。还请注意，我在那个字符串的末尾加上了一些文字，它们可以让我在踪迹文件里一眼看出哪些信息来自被我修改过的嵌入式服务器。

代码清单6-3 修改libmysqld.c文件

```
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* Adds a method to permit embedded applications to call DEBUG_PRINT */
void STDCALL mysql_debug_print(const char *a)
{
    DEBUG_PRINT(a, (" -- Embedded application. "));
}
/* END CAB MODIFICATION */
```

如果是在Windows环境下给嵌入式程序库添加一个新方法，还需要修改libmysqld.def文件把新方法包括进去。在代码清单6-4给出的代码片段里，我已经把mysql_debug_print()语句添加到这个文件里

了。请注意，这个文件是按照字母表顺序来排列那些函数的。

代码清单6-4 修改libmysqld.def文件

```
LIBRARY    LIBMYSQLD
DESCRIPTION 'MySQL 5.0 Embedded Server Library'
VERSION    5.0
EXPORTS
    _dig_vec_upper
    _dig_vec_lower
    ...
    mysql_debug_print
    mysql_debug
    mysql_dump_debug_info
    mysql_eof
    ...
```

大功告成！现在，重新编译嵌入式服务器之后，新方法就可以在应用程序里使用了。我已经在我的嵌入式服务器里这样做过。后面的例子将使用这个方法把字符串写入踪迹文件。当我需要把踪迹文件里的某条消息和MySQL源代码里的某条语句对应起来的时候，这个办法帮上了大忙。

提示 刚才那几个代码清单使用了第3章描述的注释策略。希望大家也能这样做，当需要升级到一个新版本的时候，这些注释可以让你们迅速找到曾经修改过的地方。

6.4.3 数据问题如何解决

在动手编写和运行你们的第一个嵌入式MySQL应用程序之前，应该把将要使用的数据考虑清楚。如果你打算编写的嵌入式应用程序只是用来提供一个管理接口，以便创建表和填充它们，那就无所谓数据问题。不过，如果你打算编写的不是一个这样的接口或类似的工具，那你肯定需要使用其他的工具来配置数据库，而这意味着你必须考虑数据的问题。

还好，只要你使用的是简单的表类型（比如MyISAM），就可以使用一个独立型服务器和你最喜欢的工具，来创建你的数据库和表并填充它们。一旦数据全部填充完毕，就可以把有关的子目录和文件从独立型服务器下复制到另一个地方——这么做的目的是为了把嵌入式服务器和独立型服务器的数据存放位置区分开，这一点非常重要。把那个新地点记在你的小本子里，在配置嵌入式服务器的时候还需要用到它。

我在本书的所有例子里以及我自己所使用的嵌入式应用程序里都是这样做的。这样我就可以自由地设计数据库和表并用我想使用的数据来填充它们，而无须考虑是否需要创建一个管理接口的问题。事实上，绝大多数嵌入式MySQL应用程序都是按照这个套路创建的。

6.4.4 创建基本的嵌入式服务器

前面的几个小节已经介绍了使用嵌入式库需要用到的所有函数。下面是一个简单的例子，它用了刚才介绍过的所有函数。这个例子有两个版本，一个适用于Linux平台，另一个适用于Windows平台。除了源代码中的几处细节，这两个版本几乎完全一样，最大的区别在于程序的编译过程。本章的所有例子都假设你使用的是一个在调试模型下编译出来的嵌入式库。

示例程序将完成以下几项工作：从嵌入式服务器的数据目录里读出一份数据库名单并把那份名单输出到控制台，创建一个名为testdb1的新数据库，再次读出一份数据库名单并把它输出到控制台，最后再删除testdb1数据库。虽然不是很复杂，但刚才介绍过的函数都会用上。我还特意引入了一些用来启用踪迹文件（DEBUG功能）的调用，以及使用新添加到嵌入式库里的mysql_debug_print()函数把信息写入踪迹文件的调用。

1. Linux版示例

首先，需要创建一个配置文件（my.cnf）。你可以使用一个现有的配置文件，但我建议你把它复制到嵌入式服务器的安装目录下。比如说，如果你为嵌入式服务器创建的子目录是/var/lib/mysql_embedded，应该把它的配置文件放在那里并把它所有的数据子目录（数据库文件和文件夹）也复制到那个子目录里去，而那个子目录也应该只包含这些文件。唯一的例外情况是你有两个从功能到用法都一模一样，但用来支持不同语言的嵌入式服务器；比如说一个用来支持英语，一个用来支持中文。对于这种情况，建议你把它们可以共享的文件用一个独立型服务器创建并填充好，然后集中到一个子目录；把它们不能共享的文件分开存放到不同的子目录里并通过各自的配置文件加以引用。代码清单6-5是我为这个示例程序编写的配置文件。

代码清单6-5 Linux版示例程序的配置文件（my.cnf）

```
[mysqld]
basedir=/var/lib/mysql_embedded
datadir=/var/lib/mysql_embedded
#slow query log#
#tmpdir#
#port=3306
#set-variable=key_buffer=16M

[libmysqld_client]
#debug=d:t:i:0,\\mysqld_embedded.trace
```

请注意，这个配置文件里的大部分选项都是禁用的（行首有#字符），这么做的好处是可以在需要时方便快捷地激活各有关选项。只有调试功能处于禁用状态，我才有机会向你们演示如何通过编程启用它。

接下来，还需要编写这个应用程序的源代码文件。如果仔细阅读了刚才介绍MySQL C API时给出的示例代码，你就应该很熟悉它们了。代码清单6-6是这个简单的嵌入式MySQL应用程序的完整源代码。

代码清单6-6 嵌入式应用程序示例1（Linux平台：example1_linux.c）

```
#include <my_global.h>
#include <mysql.h>

MYSQL *mysql;           //the embedded server class
MYSQL_RES *results;     //stores results from queries
MYSQL_ROW record;       //a single row in a result set

/*
```


These variables set the location of the ini file and data stores.

```

*/
static char *server_options[] = {"mysql_test",
    "--defaults-file=/var/lib/mysql_embedded/my.cnf",
    "--datadir=/var/lib/mysql_embedded" };
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmysqld_server", "libmysqld_client" };

int main(void)
{
    /*
        This section initializes the server and sets server options.
    */
    mysql_server_init(num_elements, server_options, server_groups);
    mysql = mysql_init(NULL);
    mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
    mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
    /*
        The following call turns debugging on programmatically.
        Comment out to turn off debugging.
    */
    //mysql_debug("d:t:i:0,\\mysqld_embedded.trace");
    /*
        Connect to embedded server.
    */
    mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema",
        0, NULL, 0);
    /*
        This section executes the following commands and demonstrates
        how to retrieve results from a query.

        SHOW DATABASES;
        CREATE DATABASE testdb1;
        SHOW DATABASES;
        DROP DATABASE testdb1;
    */
    mysql_debug_print("Showing databases.");           //record trace
    mysql_query(mysql, "SHOW DATABASES;");             //issue query
    results = mysql_store_result(mysql);               //get results
    printf("The following are the databases supported:\n");
    while(record=mysql_fetch_row(results))              //fetch row
    {
        printf("%s\n", record[0]);                     //process row
    }
    mysql_debug_print("Creating the database testdb1."); //record trace
    mysql_query(mysql, "CREATE DATABASE testdb1;");
    mysql_debug_print("Showing databases.");
    mysql_query(mysql, "SHOW DATABASES;");             //issue query
    results = mysql_store_result(mysql);               //get results
    printf("The following are the databases supported:\n");

```



```

while(record=mysql_fetch_row(results))           //fetch row
{
    printf("%s\n", record[0]);                    //process row
}
mysql_free_result(results);
mysql_debug_print("Dropping database testdb1."); //record trace
mysql_query(mysql, "DROP DATABASE testdb1;");    //issue query
/*
    Now close the server connection and tell server we're done (shutdown).
*/
mysql_close(mysql);
mysql_server_end();

return 0;
}

```

我在代码里加了许多注释（也许有点而多余）来帮助大家阅读它们。首先创建了几个全局变量，并设置好初始化数组。接下来用那个数组里的选项对服务器进行了初始化，又设置了几个连接选项，然后连接上了服务器。这个示例应用程序的主体部分从数据库读出数据并把它打印出来。在这个示例程序的末尾部分，我先断开了与服务器的连接，然后关闭了服务器。

编译这个示例程序需要先用mysql_config脚本把需要用到的各种库找出来。这个脚本可以把你作为参数传递给它的每一个选项的实际路径返回到命令行。从命令行直接运行这个脚本，可以查看到所有的选项和它们的值。下面是用来编译这个示例程序的命令。

```

gcc example1_linux.c -g -o example1_linux
    '/usr/local/mysql/bin/mysql_config --include --libmysqld-libs'

```

这条命令适用于绝大多数Linux平台，但偶尔也会出现它不能完成编译的问题。如果你的MySQL安装目录与我在这个示例里使用的不一样，请对mysql_config脚本的路径做相应的调整。如果你在系统上安装了多个MySQL副本，或者如果你把嵌入式库安装在了其他地方，则可能无法使用mysql_config脚本，因为它将返回错误的库路径。如果你在机器里安装了多个版本的MySQL源代码，可能也会遇到这样的问题。不管怎么说，一定要避免使用的是某个版本的头文件，但编译的却是另一个版本的嵌入式库的情况。此外，如果系统里没有之前的glibc库或是版本不匹配，也可能会遇到一些麻烦。

为了避免这些问题，应该先在命令行上直接运行一次mysql_config脚本并核对一下各个库的路径。在编写代码的时候，还要注意别把库文件和头文件的路径写错了。下面是为了避免这些问题而使用过的一条编译命令（我曾在SUSE机器上遇到过所有这些问题）。

```

g++ example1_linux.c -g -o example1_linux -lz -I/usr/include/mysql
-L/usr/lib/mysql -lmysqld -lz -lpthread -lcrypt -lnsl -lm -lpthread -lc
-lnss_files -lnss_dns -lresolv -lc -lnss_files -lnss_dns -lresolv -lrt

```

请注意，我在这条命令里使用的是比较新的g++编译器，而不是常见的gcc编译器。这是因为我的机器里只有最新的GNU库，没有相应的老版本。当然，我可以通过下载那些老版本库的办法来解决问题，但那会比使用g++编译器更费事。所以说，做程序员的都很懒。

代码清单6-7是在一个典型的MySQL系统上运行这个示例程序得到的输出。顺便说一句，这个例子使用的数据都是从一个独立型服务器的数据目录复制到嵌入式服务器的数据目录里的。

代码清单6-7 示例输出

```

linux:/home/Chuck/source/Embedded # ./example1_linux
The following are the databases supported:
information_schema
mysql
test
The following are the databases supported:
information_schema
mysql
test
testdb1
linux:/home/Chuck/source/Embedded #

```

希望读者能多花点儿时间在自己的机器上好好练习一下这个示例程序。可以把这个示例程序里的查询命令替换为其他的查询命令试试看,这可以让你对如何编写自己的嵌入式MySQL应用程序有一个直观的感受。如果你在嵌入式库里实现了mysql_debug_print()函数,还可以用这个示例程序来测试它一下,具体做法有两种:一是在源代码里删除mysql_debug()函数调用语句前的注释符号;二是在配置文件里删除debug选项前的注释符号。

下一个例子将演示如何封装嵌入式库里的各种函数,并在一个更有现实意义的示例程序里演示它们的用法。

2. Windows版示例

首先,需要创建一个配置文件(my.ini)。你可以使用一个现有的配置文件,但建议你把它复制到嵌入式服务器的安装目录下。比如说,嵌入式服务器创建的子目录是c:/mysql_embedded,就应该把它的配置文件放在那里并把所有的数据子目录(数据库文件和文件夹)也复制到那个子目录里去,而那个子目录也应该只包含这些文件。唯一的例外情况是,你有两个从功能到用法都一模一样,但用来支持不同语言的嵌入式服务器;比如一个用来支持英语,一个用来支持中文。对于这种情况,建议你把它们可以共享的文件用一个独立型服务器创建并填充好,然后集中到一个子目录,把它们不能共享的文件分开存放到不同的子目录里并通过它们各自的配置文件加以引用。代码清单6-8是为这个示例程序编写的配置文件。请注意,这个配置文件里的大部分选项都是禁用的(行首有#字符),因为我使用的是默认配置。我把那些选项以注释形式留在配置文件里,是为了让大家知道都有哪些常用的选项,以及它们都应该出现在配置文件里的什么地方。

代码清单6-8 Windows版示例程序的配置文件(my.ini)

```

[mysqld]
basedir=C:/mysql_embedded
datadir=C:/mysql_embedded/data
language=C:/mysql_embedded/share/english
#slow query log#
#tmpdir#
#port=3306
#set-variable=key_buffer=16M

```

```
[libmysqld_client]
#debug=d:t:i:0,\\mysqld_embedded.trace
```

创建项目文件稍微有点儿棘手。为了最大限度地发挥Visual Studio的能力,建议读者这样做:打开MySQL源代码根目录里的主解决方案文件,然后把新应用程序作为一个新项目添加到那个解决方案里。因为这只是一种练习,所以你可能不想把代码保存到MySQL源代码树里去;你可以把代码保存到另外一个子目录,但别忘了给它起个有意义的名字,让这个名字能帮助你在日后回忆起它们对应着哪个版本的MySQL源代码。

可以用Visual Studio的“项目向导”来创建这个项目。你应该选择C++►Win32 Console项目模板,并命名这个项目。这将在“项目向导”指定的根目录下创建一个与这个项目同名的新文件夹。你应该创建一个空项目并添加自己的源代码文件。

把一个项目文件创建为解决方案的一个子项目,可以带来许多非常酷的好处。比如说,只需把libmysqld项目添加到你的项目依赖关系里,就可以让Visual Studio自动完成对它的编译(你用不着编写任何制作文件!)。可以通过菜单命令Project ► Project Dependencies打开项目依赖关系工具。你还应该通过解决方案的Configuration下拉菜单,把编译配置设置为Active(Debug),通过解决方案的Platform下拉菜单把平台设置为Active(Win32),这两个菜单都在主工具条上。

还需要在项目属性对话框里设置几个开关。项目属性对话框可以通过菜单命令Project ► Properties打开,也可以通过右击某个项目再选择Properties打开。需要检查的第一个开关是运行库将如何生成,请把这个开关设置为Multi-threaded Debug DLL (/MDd),具体做法是:在项目属性树里展开C/C++条目,单击Code Generation子条目,然后从Runtime Library下拉列表里选中上述设置。这样的设置将使你的应用程序使用一个支持调试功能、支持多线程和DLL的运行库版本。这个选项在“项目属性”对话框里的位置如图6-1所示。

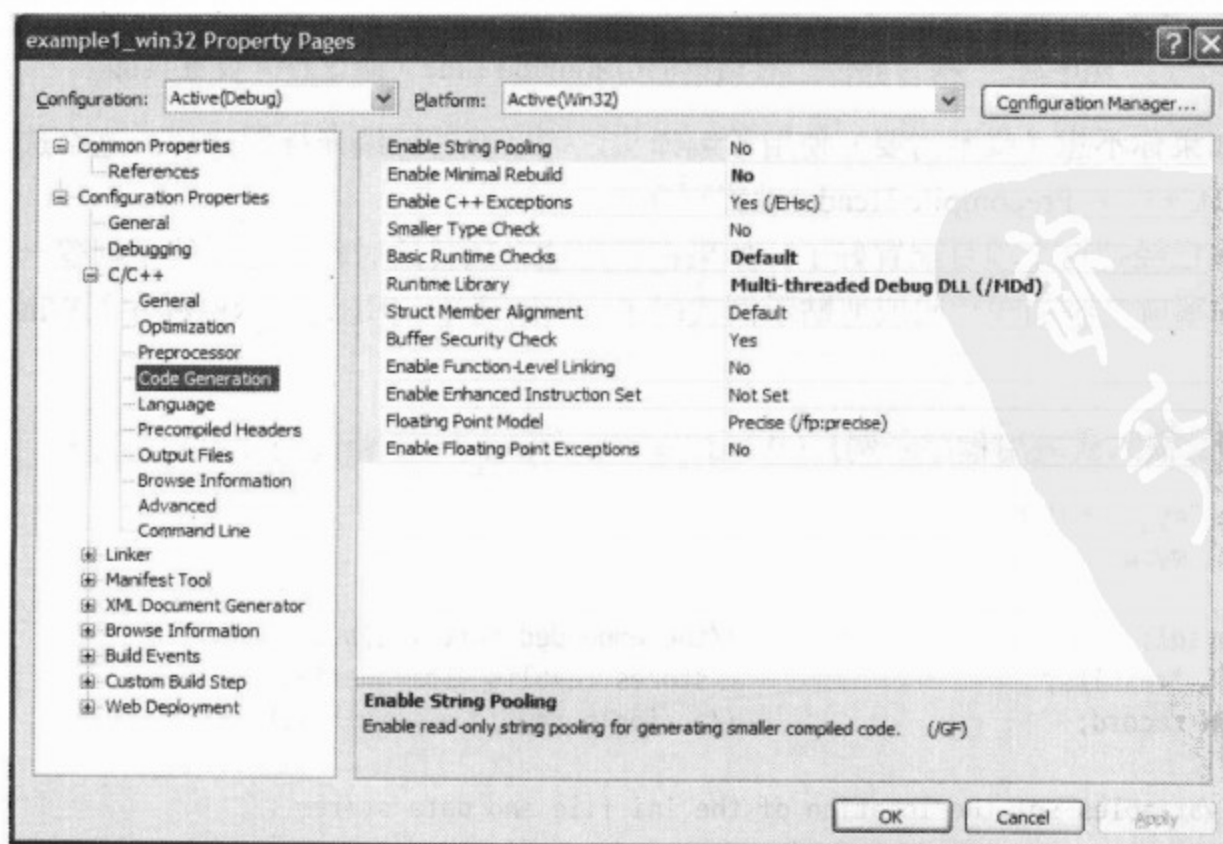


图6-1 “项目属性”对话框的Code Generation(代码生成)设置页面

需要修改的下一个属性是把MySQL头文件的目录路径添加到项目属性里。完成这个修改最简单的做法是先展开C/C++条目，然后单击Command Line子条目。你将在右边的窗口里看到一些命令行参数。如果想添加新的参数，把它输入到Additional Options文本框里就行了。具体到这个例子，请输入“/I ../include”——如果没有把项目存放在MySQL源代码树里，请对这个参数做相应的调整。这个选项在项目属性对话框里的位置如图6-2所示。

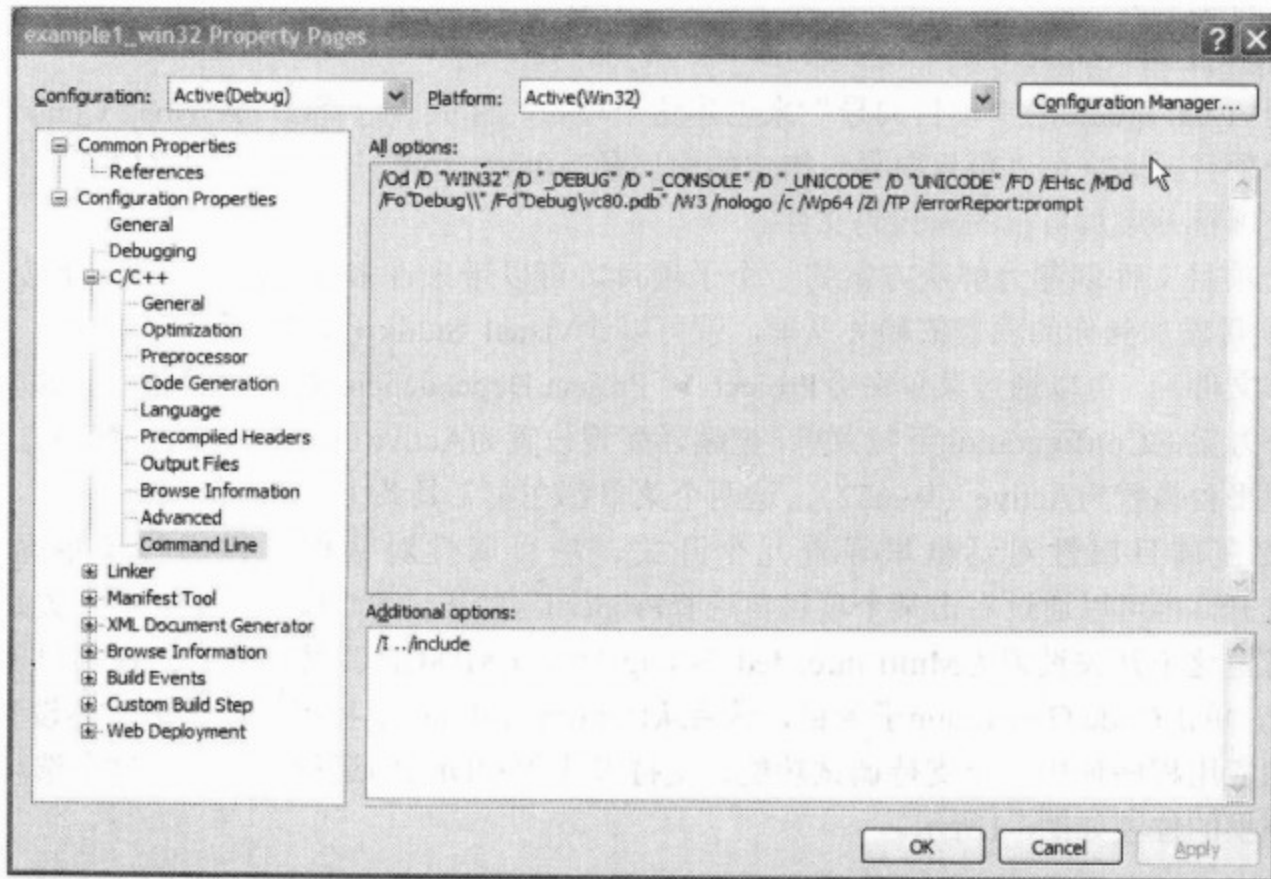


图6-2 “项目属性”对话框的Command Line（命令行）设置页面

此外，如果你不想（或不需要）使用预编译头，可以弃选预编译标题选项。这个选项在项目属性对话框里的C/C++ ➤ Precompile Header设置页面上。

现在，你已经把这个项目配置好了。如果在创建这个项目的时候，选择的是创建一个基本的项目文件，剩下的事就是添加源代码或剪贴示例代码了。代码清单6-9是这个示例程序的Windows版本的完整源代码。

代码清单6-9 嵌入式应用程序示例1（Windows平台：example1_wind32.cpp）

```
#include "my_global.h"
#include "mysql.h"

MYSQL *mysql;                //the embedded server class
MYSQL_RES *results;          //stores results from queries
MYSQL_ROW record;            //a single row in a result set
/*
  These variables set the location of the ini file and data stores.
*/
static char *server_options[] = {"mysql_test",
```



```

"--defaults-file=c:\\mysql_embedded\\my.ini",
"--datadir=c:\\mysql_embedded\\data" };
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmysqld_server", "libmysqld_client" };

int main(void)
{
    /*
     * This section initializes the server and sets server options.
     */
    mysql_server_init(num_elements, server_options, server_groups);
    mysql = mysql_init(NULL);
    mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
    mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
    /*
     * The following call turns debugging on programmatically.
     * Comment out to turn off debugging.
     */
    //mysql_debug("d:t:i:0,\\mysqld_embedded.trace");
    /*
     * Connect to embedded server.
     */
    mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema",
        0, NULL, 0);
    /*
     * This section executes the following commands and demonstrates
     * how to retrieve results from a query.
     */
    SHOW DATABASES;
    CREATE DATABASE testdb1;
    SHOW DATABASES;
    DROP DATABASE testdb1;
    /*
    mysql_debug_print("Showing databases.");           //record trace
    mysql_query(mysql, "SHOW DATABASES;");             //issue query
    results = mysql_store_result(mysql);               //get results
    printf("The following are the databases supported:\n");
    while(record=mysql_fetch_row(results))             //fetch row
    {
        printf("%s\n", record[0]);                    //process row
    }
    mysql_debug_print("Creating the database testdb1."); //record trace
    mysql_query(mysql, "CREATE DATABASE testdb1;");
    mysql_debug_print("Showing databases.");
    mysql_query(mysql, "SHOW DATABASES;");             //issue query
    results = mysql_store_result(mysql);               //get results
    printf("The following are the databases supported:\n");
    while(record=mysql_fetch_row(results))             //fetch row
    {
        printf("%s\n", record[0]);                    //process row
    }

```

```

    }
    mysql_free_result(results);
    mysql_debug_print("Dropping database testdb1.");           //record trace
    mysql_query(mysql, "DROP DATABASE testdb1;");             //issue query
    /*
    Now close the server connection and tell server we're done (shutdown).
    */
    mysql_close(mysql);
    mysql_server_end();

    return 0;
}

```

代码里加了许多注释（也许有点儿多余）来帮助大家阅读。首先创建了几个全局变量，并设置好了我的初始化数组。接下来，我用那个数组里的选项对服务器进行了初始化，又设置了几个连接选项，然后连接上了服务器。这个示例应用程序的主体部分从数据库读出数据并把它打印出来。在这个示例程序的末尾部分，我先断开了与服务器的连接，然后关闭了服务器。

编译这个示例程序非常简单，选择 **Build ► Build example1_win32** 即可。如果你已经编译过 `libmysqld` 项目，这次将只编译这个示例程序；如果 `libmysqld` 库或其他库的 `object` 文件因为某种原因不存在或过时了，**Visual Studio** 会再次编译它们。

注意 你们可能会在 `mysql_com.h` 或其他的头文件里遇到一些奇怪的错误，而导致问题的原因很可能是这样一个优化策略：**Microsoft** 在 `stdafx.h` 文件里自动加上了一条 `#define WIN32_LEAN_AND_MEAN`，如果你打开的这个开关，**Visual Studio** 的编译器（通常）会在编译和链接时省略一些头文件。如果把这行语句删掉（或改为注释）再重新编译一次，程序就应该可以通过编译了。如果你根本没有用到 `stdafx.h` 文件，就应该不会遇到这个问题。

在编译工作结束之后，读者可以通过 **Visual Studio** 的调试菜单来运行这个程序了，另一个办法是打开一个命令窗口并从命令行来运行这个程序。如果这是你第一次运行它，应该看到一条如下所示的出错消息。

```

This application has failed to start because LIBMYSQLD.dll was not found.
Re-installing the application may fix this problem.

```

这个错误的原因与出错消息里的第2句话没有任何关系，真正的根源是 `libmysqld` 库不在搜索路径上。如果你一直在与 `.NET` 或 `COM` 打交道，从没用过 `C` 语言库，可能从未见到过这种错误。与 `.NET` 或 `COM` 的情况不同，`C` 语言库没有被注册到 `GAC`（`Global Assembly Cache`，全局汇编缓存）或注册表里，所以在运行时会找不到它们。你应该把这些库（`DLL`）与你的应用程序放在同一个子目录里，至少应该把它们放在某个搜索路径上。绝大多数开发人员会把这些 `DLL` 复制到可执行文件所在的目录里去。

如果你真的遇到了这个问题，需要把 `libmysqld.dll` 文件从 `lib_debug` 子目录复制到 `example1_win32.exe` 文件所在的子目录，或是把 `lib_debug` 子目录添加到搜索路径上。在解决了这个问题之后，就应该看到如代码清单6-10所示的输出了。

代码清单6-10 示例输出

```
D:\source\C++\mysql-5.0.22\example1_win32\Debug>example1_win32
The following are the databases supported:
information_schema
cluster
mysql
test
The following are the databases supported:
information_schema
cluster
mysql
test
testdb1
```

希望读者能多花点儿时间在自己的机器上好好练习一下这个示例程序。可以把这个示例程序里的查询命令替换为其他的查询命令试试看,这可以让你对如何编写自己的嵌入式MySQL应用程序有一个直观的感受。如果你在libmysqld库里实现了mysql_debug_print()函数,还可以用这个示例程序来测试它一下,具体做法有两种:一是在源代码里删除mysql_debug()函数调用语句前的注释符号;二是在配置文件里删除debug选项前的注释。

6.4.5 出错处理问题如何解决

读者也许正想知道应该如何解决出错处理的问题。具体地说,怎样才能探测到嵌入式服务器里发生的错误并妥善地加以处理呢? libmysqld库里的许多函数在执行出错时都可以返回一个相应的出错代码,你可以捕获它们并采取必要的行动。在前面介绍各有关函数的时候,已经对它们的返回值情况进行了描述。虽然我在第一个嵌入式MySQL应用程序示例里没有进行什么出错处理,但将在下一个示例里进行出错处理。请注意我是如何捕获那些错误并把它们发送到客户端程序去的。

6.4.6 嵌入式服务器应用程序

前面的例子已向大家演示了如何编写一个基本的嵌入式MySQL应用程序。虽然那些例子演示了如何建立连接并从一个专用的MySQL安装那里读取数据,但它们解决的只是些最基本的底层问题(连出错处理都没有!),如果你想编写一个有实用价值的嵌入式应用程序,它们的帮助恐怕不会有多大。这一节的例子就不一样了,虽然场景也是虚构的,但它将把编写一个有实用意义的嵌入式应用软件所需要的工具提供给大家。

这个应用软件的名字是Book Vending Machine(中文意思是“图书销售机”,以下简称BVM),它是一个运行在带有触摸屏的基于Microsoft Windows的专用个人电脑上的、嵌入式系统,如图6-3所示。这个系统以及它的其他输入设备都安装在一台专用的自动售货机里,它售卖的东西是图书。BVM可以让出版公司把最畅销的图书放在一个半移动式设备里进行销售,并根据具体情况配置和补充图书。出版公司可以把BVM安放在展览会、飞机场、购物中心等寸土寸金的场所。这类场所的客流量很大,有兴趣购买图书的人应该不少。因为占地不大,也不需要售货员,BVM可以让出版公司省下许多成本。

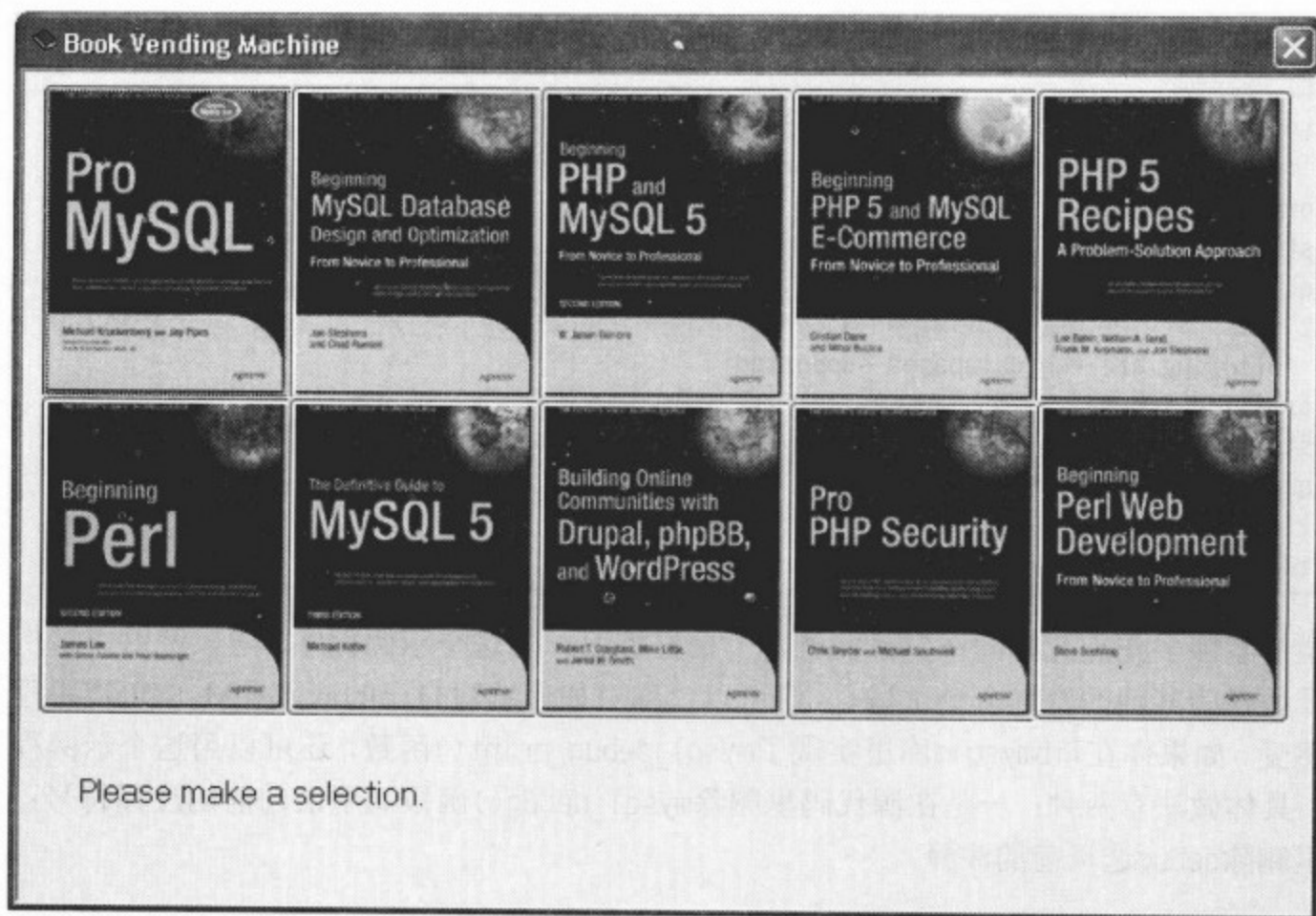


图6-3 BVM自动图书销售机的顾客操作界面

注解 我经常会感到困惑：人们对这个话题是不是真的关心？我曾见过好几篇文章谈到“按需出版”（print-on-demand）将成为一种未来的潮流，但我却从来没有见过有书或文章分析图书销售机的工作原理。我知道有几家出版公司已经推出了几种概念性产品，但可惜的是，它们都没能引起多大的关注。我选用这个例子呼吁大家关注现实。我自己在阅读技术书刊的时候，就很反感那些毫无实际用途或无足轻重的例子，希望你认可我这个例子至少是言之有物的。

1. 操作界面

BVM需要有两个操作界面：一个负责完成正常的图书销售活动，另一个让出版公司补充图书并对有关信息进行必要的调整。按照我的设想，BVM应该把机器里现有图书的封面缩微图以按钮的形式显示在一块触摸屏上供顾客选择。现如今的自动售货机大都通过一组发光按钮来操作，如果某种东西还没卖完，与之对应的按钮是发光的，如果某种东西卖完了，与之相对应的按钮将不发光。我决定让BVM也具备这种行为，根据某种图书是否已经卖完而让触摸屏上与之对应的按钮闪亮或是变暗。

当顾客在触摸屏上按下某种图书的按钮时，屏幕上将会出现那本书的详细介绍和价格。如果顾客决定购买那本书，他可以单击“购买”按钮并按照屏幕提示完成付款和取书操作。本节的示例程序将模拟上述这些活动。如果真的有像BVM这样的产品，它应该还有一些硬件设备来完成收款、验证付款信息、从机器里取出图书并交付给顾客等一系列硬件动作。图6-5是BVM的主操作界面，图6-4是它在部分图书缺货时的样子。图6-5是顾客按下某个按钮后看到的关于那本书的详细信息。

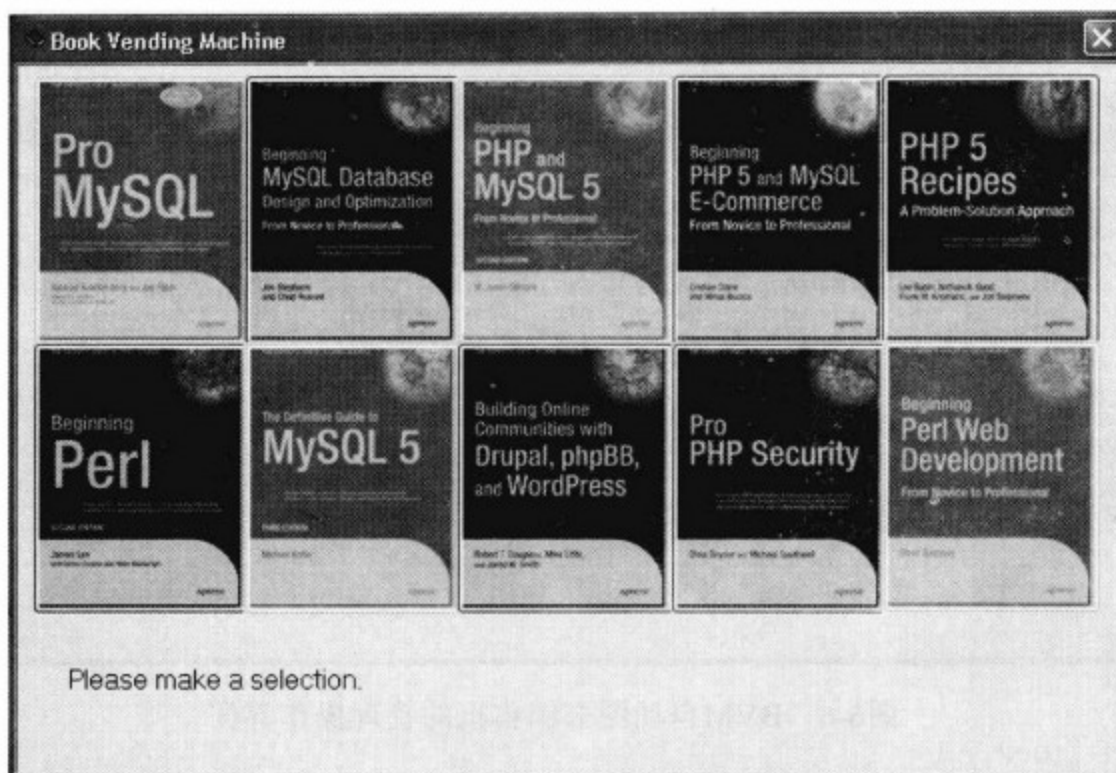


图6-4 BVM自动图书销售机的顾客操作界面在部分图书缺货时的样子

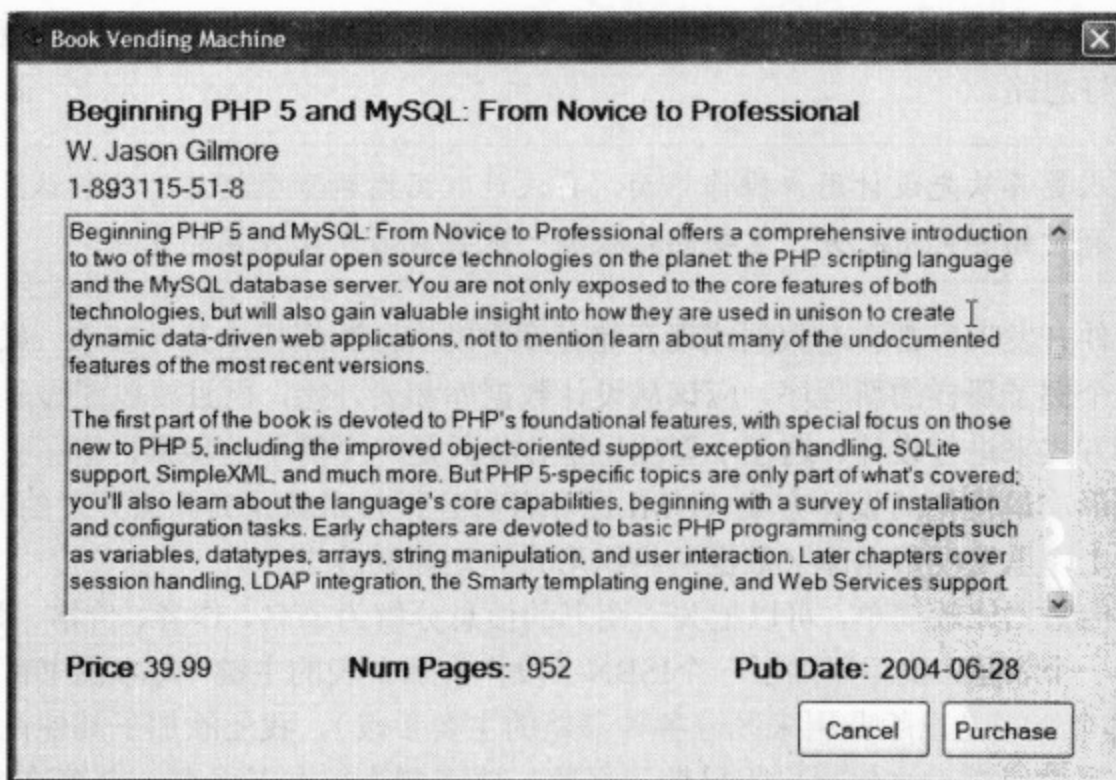


图6-5 BVM自动图书销售机的图书介绍画面

自动售货机必须能补充货物, 否则就会没有意义。BVM是通过一个管理操作界面实现这一功能的。在需要补充/更换图书或修改某本书的详细介绍时, 出版公司的人员可以打开机器并关闭这个嵌入式应用程序(这个功能需要添加到我们的例子里), 然后再用如下所示的命令重新启动这个应用程序进入管理操作界面:

```
C:\>Books BookVendingMachine -admin
```

出版公司的人员可以通过管理操作界面进行各种必要的查询和修改。BVM图书销售机的管理操作

界面如图6-6所示，可以在图中看到一条典型的用来修改图书库存数量的数据库命令。

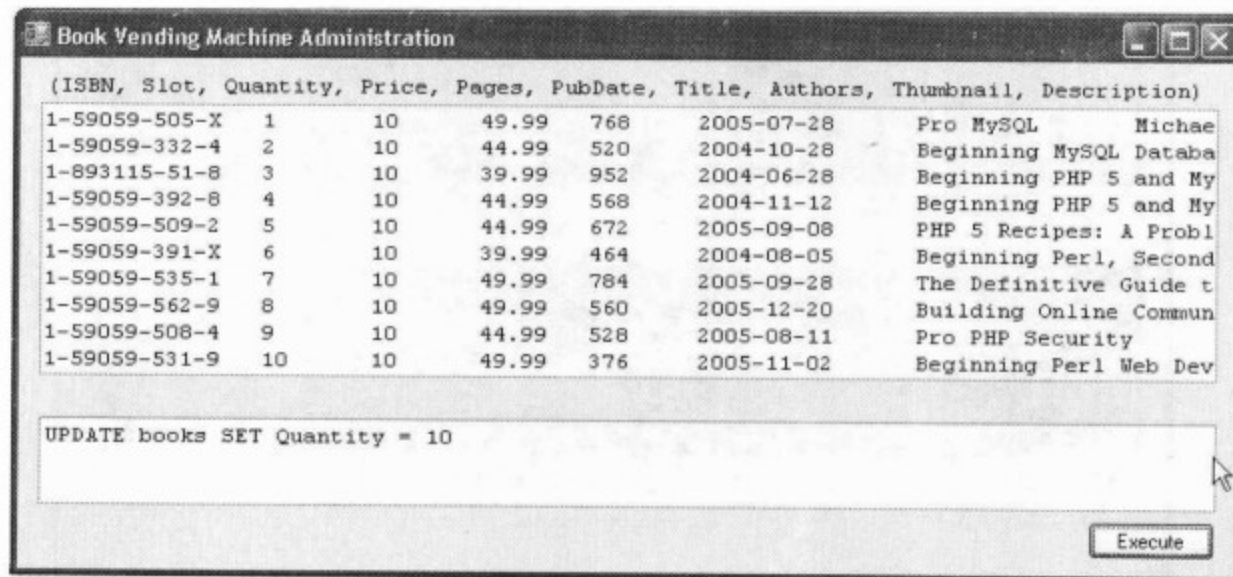


图6-6 BVM自动图书销售机的管理操作界面

2. 数据和数据库

这个例子所使用的数据，是先用一个独立型MySQL服务器创建好以后，再复制到这个嵌入式MySQL服务器的数据目录里的。在创建这个应用程序的时候，最先完成的是数据结构和数据库的设计。这是大家公认的好思路。

注解 有些开发人员喜欢先设计用户操作界面，再设计数据结构和数据库，他们认为那种做法更好。这里并没有对错和优劣之分，重要的是数据一定要成为设计方案的重点。

绝大多数软件开发项目都会对数据或现存储存库中的实际数据提出某种要求。如果你接受的任务是开发一个像这个例子那样的新程序，应该从设计数据库和表开始，而且要以能显示各项以及各项之间的内在联系的方式来进行设计。这在小型项目里往往只是一个简单的步骤，但在大型项目里往往需要反复多次才能形成最优化的设计方案（UML图和各种建模技术就是为此而开发的）。有时候，你们甚至会在后面的工作里发现数据库（数据的组织形式）还需要改进。

这个例子中的数据比较简单，可以把关于图书的信息分解为书名、作者、价格、图书简介等字段，并把它们保存到一个表里。我还添加了一个ISBN字段作为那个表的主键（每本图书都有一个独一无二的ISBN编码，这个编码是出版业用来区分各种书籍的主要手段）。我还添加了其他几个字段来存放选购图书时希望看到的信息，这包括出版日期和页数。还需要为每本书保存一个缩略图。（我的选择是把这些缩略图保存为一些外部文件，在需要时通过文件系统读取它们，在数据库里只保存它们的路径和文件名。我认为这么做比较简单，虽然更容易出问题。另一种选择是用一个BLOB字段把那些缩略图保存到数据库的内部。）最后，我分析了运行用户界面所需的東西并决定再添加两个字段：一个用来记录那些图书在BVM机器里的存放槽编号，另一个用来记录图书的存货数量。我把这个表命名为books，并把它放到了一个名为bvm的数据库里。下面是我用来创建bvm数据库和books表的CREATE语句。代码清单6-11是用EXPLAIN命令查看到的books表的布局结构。

```
CREATE DATABASE BVM;
CREATE TABLE Books (ISBN varchar(15) NOT NULL,
```

```
Title varchar(125) NOT NULL, Authors varchar(100) NOT NULL,
Price float NOT NULL, Pages int NOT NULL, PubDate date NOT NULL,
Quantity int DEFAULT 0, Slot int NOT NULL, Thumbnail varchar(100) NOT NULL,
Description text NOT NULL);
```

代码清单6-11 Books表的结构

```
mysql> explain Books;
```

Field	Type	Null	Key	Default	Extra
ISBN	varchar(15)	NO			
Title	varchar(125)	NO			
Authors	varchar(100)	NO			
Price	float	NO			
Pages	int(11)	NO			
PubDate	date	NO			
Quantity	int(11)	YES		0	
Slot	int(11)	NO			
Thumbnail	varchar(100)	NO			
Description	text	NO			

10 rows in set (0.08 sec)

为了管理那些缩略图，我决定只把缩略图的文件名存放在thumbnail字段，然后用一个系统级配置选项来给出它们的路径。做这件事情的一种办法是创建一个命令行开关，另一种办法是把路径信息保存到MySQL配置文件里并从那里读出它。还可以从数据库里读出它。我选择的办法是创建一个名为settings的表，这个表只包含两个字段：一个是FieldName，用来存放选项的名字（如ImagePath）；另一个是Value，用来存放选项的值（如c:\images\mypic.tif）。这个办法的好处是我可以创建任意多个系统选项并从外部控制它们。下面是我用来创建settings表的CREATE SQL命令，还用一条INSERT命令为这个示例应用程序设置好了ImagePath选项：

```
CREATE TABLE settings (FieldName varchar(20), Value varchar(255));
INSERT INTO settings VALUES ("ImagePath", "c:\\mysql_embedded\\images\\");
```

3. 创建项目

创建这个项目的最佳办法是用Visual Studio的“项目向导”创建一个新的Windows项目。我建议你们打开MySQL源代码根目录下的主解决方案文件，并把新程序添加为那个解决方案的一个新项目。因为这只是一种练习，所以你可能不想把代码保存到MySQL源代码树里去；你可以把代码保存到另外一个子目录，但别忘了给它起个能帮助你在日后回忆起它们对应着MySQL源代码的哪个版本的名字。

可以用Visual Studio的“项目向导”来创建这个项目。应该选择CLR Windows Forms Application项目模板并命名这个项目。这将在“项目向导”指定的根目录下创建一个与这个项目同名的新文件夹。

把一个项目文件创建为解决方案的一个子项目，可以带来许多非常酷的好处。比如说，你只须把libmysqld项目添加到你的项目依赖关系里，就可以让Visual Studio自动完成对它的编译（用不着编写

任何制作文件!)。可以通过菜单命令**Project > Project Dependencies**打开项目依赖关系工具。还应该通过解决方案的**Configuration**下拉菜单把编译配置设置为**Active (Debug)**, 通过解决方案的**Platform**下拉菜单把平台设置为**Active (Win32)**, 这两个菜单都在主工具条上。

你还需要在项目属性对话框里设置几个开关。项目属性对话框可以通过菜单命令**Project > Properties**打开, 也可以通过右击某个项目再选择**Properties**打开。需要检查的第一个开关是运行库将如何生成, 请把这个开关设置为**Multi-threaded Debug DLL (/MDd)**, 具体做法是: 在项目属性树里展开**C/C++**标签, 单击**Code Generation**子条目, 然后从**Runtime Library**下拉列表里选中上述设置。这个选项在项目属性对话框里的位置如图6-1所示。

需要你修改的下一个属性是把MySQL头文件的目录路径添加到项目属性里。完成这个修改最简单的做法是先展开**C/C++**条目, 然后单击**Command Line**子条目。你将在右边的窗口里看到一些命令行参数。如果你想添加新的参数, 把它输入到**Additinal Options**文本框里就行了。具体到这个例子, 请输入**/I ../include**。如果你没有把项目存放在MySQL源代码树里, 请对这个参数做相应的调整。这个选项在项目属性对话框里的位置如图6-2所示。

此外, 如果你不想(或不需要)使用预编译标题, 可以弃选预编译标题选项。这个选项在项目属性对话框里的**C/C++ > Precompile Header**设置页面上。

最后, 还需要把**Common Language Runtime support**选项设置为**/clr**。在项目属性对话框里单击树中的**General**, 然后把**Common Language Runtime support**选项设置为**Common Language Runtime support (/clr)**。这个选项在项目属性对话框里的位置如图6-7所示。

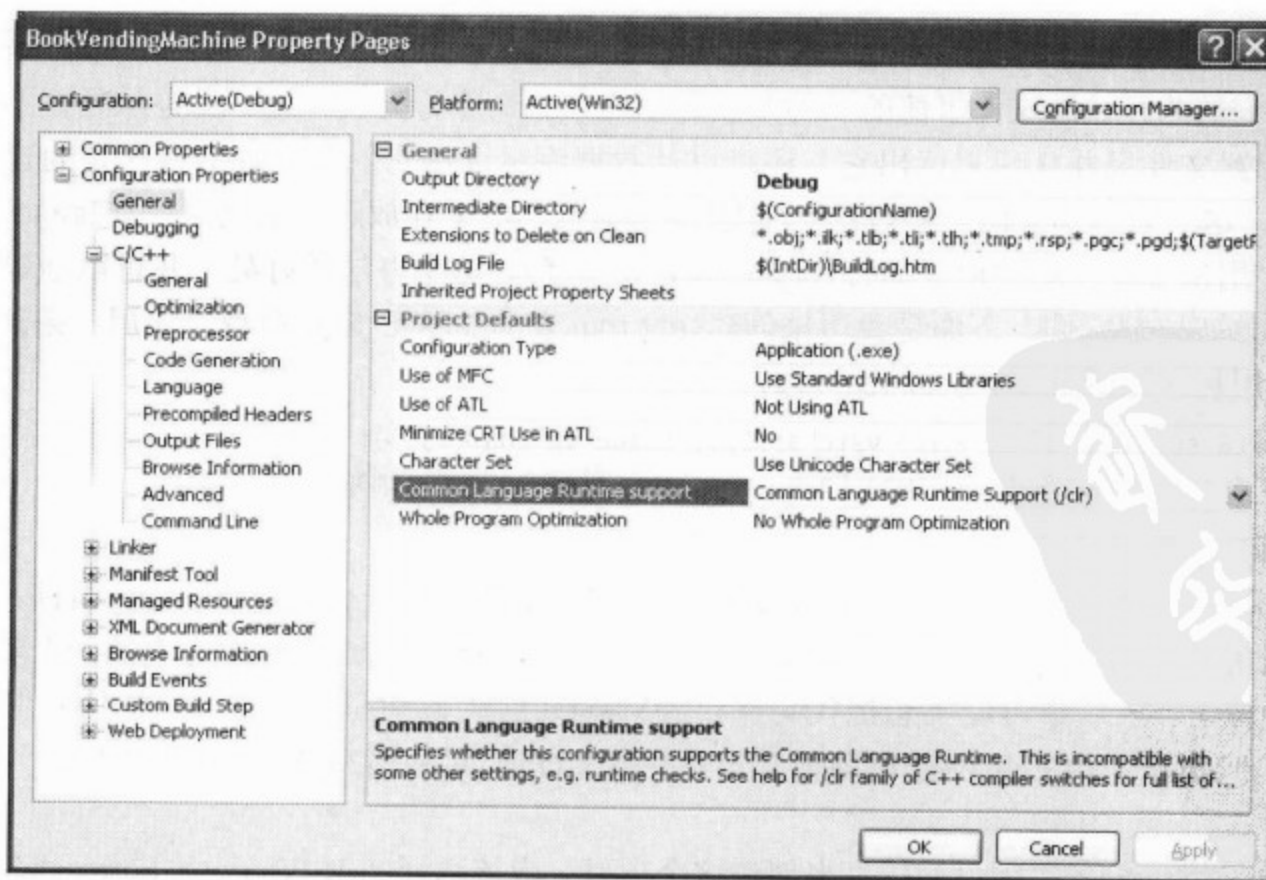


图6-7 “项目属性”对话框的General(常规)设置页面

4. 整体设计思路

这个应用程序的设计方案必须满足两个重要的要求: 一是用户界面必须易于使用并且没有任何错

误；二是必须能让我从一个.NET程序调用一个C API——如果你们到MySQL论坛和邮件表上去看看，就会发现许多人在为这个问题伤脑筋。不过，只要你们按照我这个例子里演示的办法去做，就应该不会遇到那些问题。这个问题的难点是，没有什么好办法让我们从一个.NET程序去调用嵌入式程序库里的C API函数。我解决这个问题的办法是用受控C++代码来编写程序。我知道，在一个受控应用程序里是不能使用C API调用的，但C++允许用#pragma unmanaged和#pragma managed指令暂时禁用、再重新激活这一限制。

需要调用非托管代码是程序员对库函数调用进行封装的主要动机之一。程序员利用非托管代码编写出来的DLL，能够在那些不是在.NET环境下开发的程序里使用。具体到这个例子，我将在#pragma unmanaged和#pragma managed指令之间插入一些C API调用，然后再用一个C++类把它们封装起来。这样一来，就可以让你们看到一个从.NET应用程序直接调用libmysqld库里的C API函数的例子了，够酷吧？

我还想让用户操作界面与libmysqld库完全隔离开。这么做的原因是想封装出一个功能完备的数据库访问类并把它提供给大家，作为读者编写应用程序时的基础。具体到这个例子，它还可以让我用最少的源代码——我知道，没人喜欢阅读长篇大段的源代码——把一个有现实意义的Windows应用程序呈现在读者眼前。因此，我决定把这个示例程序的数据访问部分单独设计成一个非受控C++类，并把需要用到的、来自libmysqld库的C API调用封装在这个类里。整个设计方案还包括两个表单：一个用来提供用户操作界面（名为Customer）；另一个用来提供管理操作界面（名为Administrator）。

托管代码与非托管代码

所谓“托管代码”（managed code）指的是在CLR（common language runtime，通用语言运行时）控制下运行的.NET应用程序。这些应用程序可以充分享受CLR提供的所有功能和服务，如垃圾（废弃内存）回收、更好的程序执行控制等。“非托管代码”（unmanaged code）指的是不在CLR控制下运行的Windows应用程序，它们享受不到CLR提供的各项功能和服务。

5. 数据库引擎类

在为这个示例程序设计数据库引擎类的时候，我先用纸和笔把需要用到的方法（对嵌入式MySQL服务器进行初始化、与它建立连接、以及将它关闭等）列成了一份清单。我本可以使用一个UML图分析软件来做这件事，但这个类并不复杂，所以没有那样做。这些方法很容易封装，因为它们需要的输入参数都不来自表单。

我遇到的第一个挑战是出错处理。刚才说过，我想让用户界面与libmysqld库完全隔离开，不让客户端程序知道任何关于嵌入式库的事情，在此情况下，怎样才能把出错消息发送给客户端窗体呢？可供选择的办法有很多，我选择的是实现一个出错检查方法，让客户端程序在完成一个操作后检查有没有发生错误，如果有，再调用另外一个方法去检索出错消息。这使我可以把数据库访问和窗体隔离开。

与发出查询命令和检索查询结果有关的类方法也是一种可以选择的方法，我选择的是实现一个访问迭代器，让客户端程序发出查询命令并遍历查询结果。我还需要一个用来告诉数据库“有人买走了一本书”的方法，以便数据库减少图书的存货数量。

数据检索部分包括3个方法，它们分别用来读取一个字符串字段、一个整数字段或一段文本字段。还添加了几个辅助方法来完成以下操作：从settings表读出一个设置，从数据库读出一个字段（供管

理员界面使用), 快速检索图书的存货数量。

代码清单6-12给出了这个数据库引擎类的头文件的全部源代码, 我把这个类命名为DBEngine。表6-4对这个类里的方法进行汇总。

代码清单6-12 数据库引擎类的头文件 (DBEngine.h)

```
#pragma once
#pragma unmanaged
#include <stdio.h>

class DBEngine
{
private:
    bool mysqlError;
public:
    DBEngine(void);
    const char *GetError();
    int Error();
    void Initialize();
    void Shutdown();
    char *GetSetting(char *Field);
    char *GetBookFieldStr(int Slot, char *Field);
    char *GetBookFieldText(int Slot, char *Field);
    int GetBookFieldInt(int Slot, char *Field);
    int GetQty(int Slot);
    void VendBook(char *ISBN);
    void StartQuery(char *QueryStatement);
    void RunQuery(char *QueryStatement);
    int GetNext();
    char *GetField(int fldNum);
    ~DBEngine(void);
};
#pragma managed
```

表6-4 数据库引擎类的方法

开 关	返 回 值	说 明
GetError()	char *	返回最近一次发生的错误的出错消息
Error()	int	如果服务器探测到一个出错条件, 返回1
Initialize()	void	封装着嵌入式服务器的初始化和连接操作
Shutdown()	void	返回指定参数的值; 从settings表读出信息
GetSetting()	char *	根据给定的图书槽从books表返回一个字符串值
GetBookFieldStr()	char *	根据给定的图书槽从books表返回一个字符串值
GetBookFieldText()	char *	根据给定的图书槽从books表返回一个整数值
GetBookFieldInt()	int	根据给定的图书槽从books表返回一个字符串值
GetQty()	int	根据给定的图书槽返回存货数量
VendBook()	void	根据给定的图书槽减少图书存货数量

(续)

开 关	返 回 值	说 明
StartQuery()	void	对查询迭代器进行初始化；执行查询并检索结果集
RunQuery()	void	一个辅助方法，用来执行不返回结果的查询命令
GetNext()	int	检索结果集里的下一条记录。如果已经到达结果集的末尾，返回0；如果执行成功，返回一个非零值
GetField	char *	根据给定的字段编号返回该字段的名称

在DBEngine.h文件里对数据库引擎类做出定义之后，我们还需要把那些方法实现出来。这次用不着从零开始编程了，第一个例子里的许多代码稍微修改一下，就能用在这个数据库引擎类的源代码里。代码清单6-13给出了这个数据库引擎类完整的源代码。请注意，这里使用的全局变量和用来设置初始化和启动选项的字符数组，与前面那个例子是一样的，这个部分你们应该很眼熟。请花些时间阅读这段代码，等你们看完后，我将解释几个细节问题。

代码清单6-13 数据库引擎类 (DBEngine.cpp)

```
#pragma unmanaged

#include "DBEngine.h"
#include <stdlib.h>
#include <stdio.h>
#include "my_global.h"
#include "mysql.h"

MYSQL *mysql;           //the embedded server class
MYSQL_RES *results;     //stores results from queries
MYSQL_ROW record;       //a single row in a result set
bool IteratorStarted;   //used to control iterator
MYSQL_RES *ExecQuery(char *Query);

/*
   These variables set the location of the ini file and data stores.
*/
static char *server_options[] = {"mysql_test",
    "--defaults-file=c:\\mysql_embedded\\my.ini",
    "--datadir=c:\\mysql_embedded\\data" };
int num_elements=sizeof(server_options) / sizeof(char *);
static char *server_groups[] = {"libmyswld_server", "libmysqld_client" };

DBEngine::DBEngine(void)
{
    mysqlError = false;
}

DBEngine::~DBEngine(void)
{
}
```

```
const char *DBEngine::GetError()
{
    return (mysql_error(mysql));
    mysqlError = false;
}

bool DBEngine::Error()
{
    return(mysqlError);
}

char *DBEngine::GetBookFieldStr(int Slot, char *Field)
{
    char *istr = new char[10];
    char *str = new char[128];

    _itoa_s(Slot, istr, 10, 10);
    strcpy_s(str, 128, "SELECT ");
    strcat_s(str, 128, Field);
    strcat_s(str, 128, " FROM books WHERE Slot = ");
    strcat_s(str, 128, istr);
    mysqlError = false;
    results=ExecQuery(str);
    strcpy_s(str, 128, "");
    if (results)
    {
        mysqlError = false;
        record=mysql_fetch_row(results);
        if(record)
        {
            strcpy_s(str, 128, record[0]);
        }
        else
        {
            mysqlError = true;
        }
    }
    return (str);
}

char *DBEngine::GetBookFieldText(int Slot, char *Field)
{
    char *istr = new char[10];
    char *str = new char[128];

    _itoa_s(Slot, istr, 10, 10);
    strcpy_s(str, 128, "SELECT ");
    strcat_s(str, 128, Field);
    strcat_s(str, 128, " FROM books WHERE Slot = ");
    strcat_s(str, 128, istr);
    mysqlError = false;
```



```
results=ExecQuery(str);
delete str;
if (results)
{
    mysqlError = false;
    record=mysql_fetch_row(results);
    if(record)
    {
        return (record[0]);
    }
    else
    {
        mysqlError = true;
    }
}
return ("");
}

int DBEngine::GetBookFieldInt(int Slot, char *Field)
{
    char *istr = new char[10];
    char *str = new char[128];
    int qty = 0;

    _itoa_s(Slot, istr, 10, 10);
    strcpy_s(str, 128, "SELECT ");
    strcat_s(str, 128, Field);
    strcat_s(str, 128, " FROM books WHERE Slot = ");
    strcat_s(str, 128, istr);
    results=ExecQuery(str);
    if (results)
    {
        record=mysql_fetch_row(results);
        if(record)
        {
            qty = atoi(record[0]);
        }
        else
        {
            mysqlError = true;
        }
    }
    delete str;
    return (qty);
}

void DBEngine::VendBook(char *ISBN)
{
    char *str = new char[128];
```



```
char *istr = new char[10];
int qty = 0;

strcpy_s(str, 128, "SELECT Quantity FROM books WHERE ISBN = '");
strcat_s(str, 128, ISBN);
strcat_s(str, 128, "'");
results=ExecQuery(str);
record=mysql_fetch_row(results);
if (record)
{
    qty = atoi(record[0]);
    if (qty >= 1)
    {
        _itoa_s(qty - 1, istr, 10, 10);
        strcpy_s(str, 128, "UPDATE books SET Quantity = ");
        strcat_s(str, 128, istr);
        strcat_s(str, 128, " WHERE ISBN = '");
        strcat_s(str, 128, ISBN);
        strcat_s(str, 128, "'");
        results=ExecQuery(str);
    }
}
else
{
    mysqlError = true;
}
}

void DBEngine::Initialize()
{
    /*
    This section initializes the server and sets server options.
    */
    mysql_server_init(num_elements, server_options, server_groups);
    mysql = mysql_init(NULL);
    if (mysql)
    {
        mysql_options(mysql, MYSQL_READ_DEFAULT_GROUP, "libmysqld_client");
        mysql_options(mysql, MYSQL_OPT_USE_EMBEDDED_CONNECTION, NULL);
        /*
        The following call turns debugging on programmatically.
        Comment out to turn off debugging.
        */
        //mysql_debug("d:t:i:0,\\mysqld_embedded.trace");
        /*
        Connect to embedded server.
        */
        if(mysql_real_connect(mysql, NULL, NULL, NULL, "information_schema",
            0, NULL, 0) == NULL)
        {
```

```
        mysqlError = true;
    }
    else
    {
        mysql_query(mysql, "use BVM;");
    }
}
else
{
    mysqlError = true;
}
IteratorStarted = false;
}

void DBEngine::Shutdown()
{
    /*
     * Now close the server connection and tell server we're done (shutdown).
     */
    mysql_close(mysql);
    mysql_server_end();
}

char *DBEngine::GetSetting(char *Field)
{
    char *str = new char[128];
    strcpy_s(str, 128, "SELECT * FROM settings WHERE FieldName = '");
    strcat_s(str, 128, Field);
    strcat_s(str, 128, "'");
    results=ExecQuery(str);
    strcpy_s(str, 128, "");
    if (results)
    {
        record=mysql_fetch_row(results);
        if (record)
        {
            strcpy_s(str, 128, record[1]);
        }
    }
    else
    {
        mysqlError = true;
    }
    return (str);
}

void DBEngine::StartQuery(char *QueryStatement)
{
    if (!IteratorStarted)
    {
        results=ExecQuery(QueryStatement);
    }
}
```

```
        if (results)
        {
            record=mysql_fetch_row(results);
        }
    }
    IteratorStarted=true;
}

void DBEngine::RunQuery(char *QueryString)
{
    results=ExecQuery(QueryStatement);
    if (results)
    {
        record=mysql_fetch_row(results);
        if(!record)
        {
            mysqlError = true;
        }
    }
}

int DBEngine::GetNext()
{
    //if EOF then no more records
    IteratorStarted=false;
    record=mysql_fetch_row(results);
    if (record)
    {
        return (1);
    }
    else
    {
        return (0);
    }
}

char *DBEngine::GetField(int fldNum)
{
    if (record)
    {
        return (record[fldNum]);
    }
    else
    {
        return ("");
    }
}

MYSQL_RES *ExecQuery(char *Query)
{

```



```

mysql_debug_print("ExecQuery.");
mysql_free_result(results);
mysql_query(mysql, Query);
return (mysql_store_result(mysql));
}
#pragma managed

```

在阅读上面这些代码的时候,你会看到许多与出错处理有关的语句,它们使得这个程序更加健壮。虽然我没能把所有可能的出错处理器实现出来,但最重要的出错处理功能一个也不少。

那些名字以Get开头的函数都是按照同样的套路实现的:先生成一条相应的查询命令(在客户端看不到SQL语句),执行这个查询,取回结果集并对结果集里的记录进行处理,返回一个返回值。

请大家再仔细阅读一下VendBook()方法的代码。我在这个方法里也生成了一条查询命令,但我没有取回结果集,因为那个查询命令没有结果集。不过,它还是有一个执行结果的——受影响记录的个数;如果你想进行一些额外的处理,或是在你的应用程序里进行一些规则检查,这个数字也许能派上用场。

其他的方法应该不会让读者感到陌生,它们都是我从前面的例子里复制来的,但这次给它们添加了一些与出错处理有关的语句。现在,一起去看看用户界面部分的代码是如何调用该数据库类的。

(1) 顾客界面(主窗体)

顾客界面的源代码相当长,这是因为Visual Studio会在form.h文件里自动生成许多代码。我在这里将只讨论由我本人编写的代码。希望这一节的内容能让你们了解应该如何编写自己的.NET(或其他)用户界面。除了按钮事件里的代码,我只使用了4个方法就完成了这个示例应用程序的用户界面。它们当中的第一个是DisplayError()方法,它的定义如下所示。

```
void DisplayError()
```

这个函数的用途是检查数据库类里是否发生了错误;如果是,它将把出错消息显示给用户。这个方法的具体实现是一个典型的MessageBox::Show()调用。

第二个方法是一个辅助方法,它负责把图书简介显示给用户。这个函数的名字是LoadDetails()。我对这个函数进行了抽象化,因为我知道自己需要为触摸屏上的10个按钮重复这些代码^①,对这个函数进行抽象化可以让我少写一些代码,也可以让调试工作变得容易一些。下面是这个方法的定义。

```
void LoadDetails(int Slot)
```

这个方法的输入参数是一个图书槽编号(与一个按钮编号相对应),它将调用数据库类里的方法来查询数据库并详细填充界面元素。这个方法是用用户界面与DBEngine类进行通信最多的地方。

注解 在LoadDetails()方法里会看到许多gcnew Strings(...)调用,这些代码是干什么用的?是这样的, .NET里的String类与C语言里的字符串在格式上不兼容,那些“多余”的代码用来对字符串进行必要的格式转换。

第三个方法是一个名为Delay()的辅助方法,下面是它的定义。

```
void Delay(int secs)
```

① 我发现Visual Basic只有一个功能非常酷,那就是控件数组。哦,这都是过去才有的。

这个函数的用途是产生几秒钟（由secs参数给定）的延时。我添加这个函数的目的是想用它产生的延时来模拟顾客的购书过程。如果你们的项目不像这个例子里那么“假”，就用不着把它包括在你们的源代码里。这个函数是一个用“假”代码来模拟现实操作过程的典型例子。“假”代码在设计某种操作界面的时候很有用。

第四个方法是CheckAvailability()，它的用途是根据图书的存货情况让触摸屏上的按钮发光或是不发光。下面是这个函数的定义。

```
void CheckAvailability()
```

这个函数将向数据库引擎发出一系列调用以检查每种图书的存货数量。如果某种图书买完了(quantity==0)，触摸屏上的相应按钮将从发光（可用）变成不发光（不可用）。

代码清单6-14是用户操作界面的源代码，我删去了许多由Visual Studio自动生成的代码（在代码里用“...”表示）。请注意，在这份清单的第2行就用#include "DBEngine.h"指令引入了DBEngine类的头文件，稍后几行又定义了一个DBEngine类型的变量。这个对象串起了所有的代码。因为它对窗体来说是本地的，所以我可以在任何一个事件或方法里使用它。代码里的...代表着被我删去的自动生成代码和注释。

代码清单6-14 主窗体的源代码 (MainForm.h)

```
#pragma once
#include "DBEngine.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "vcclr.h"
#include <time.h>
namespace BookVendingMachine {

    const char GREETING[] = "Please make a selection.";

    DBEngine *Database = new DBEngine();

    ...

#pragma endregion
    void DisplayError()
    {
        String ^str = gcnew String("There was an error with the database system.\n" \
                                   "Please contact product support.\nError = ");
        str = str + gcnew String(Database->GetError());
        MessageBox::Show(str, "Internal System Error", MessageBoxButtons::OK,
                          MessageBoxIcon::Information);
    }

    void LoadDetails(int Slot)
    {
        int Qty = Database->GetBookFieldInt(Slot, "Quantity");
```

```

if (Database->Error()) DisplayError();
pnlButtons->Visible = false;
pnlDetail->Visible = true;
lblStatus->Visible = false;
lblTitle->Text = gcnew String(Database->GetBookFieldStr(Slot, "Title"));
if (Database->Error()) DisplayError();
lblAuthors->Text =
    gcnew String(Database->GetBookFieldStr(Slot, "Authors"));
if (Database->Error()) DisplayError();
lblISBN->Text = gcnew String(Database->GetBookFieldStr(Slot, "ISBN"));
if (Database->Error()) DisplayError();
txtDescription->Text =
    gcnew String(Database->GetBookFieldText(Slot, "Description"));
if (Database->Error()) DisplayError();
lblPrice->Text = gcnew String(Database->GetBookFieldStr(Slot, "Price"));
if (Database->Error()) DisplayError();
lblNumPages->Text =
    gcnew String(Database->GetBookFieldStr(Slot, "Pages"));
if (Database->Error()) DisplayError();
lblPubDate->Text =
    gcnew String(Database->GetBookFieldStr(Slot, "PubDate"));
if (Database->Error()) DisplayError();
if(Qty < 1)
{
    btnPurchase->Enabled = false;
}
}
void CheckAvailability()
{
    btnBook1->Enabled = (Database->GetBookFieldInt(1, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook2->Enabled = (Database->GetBookFieldInt(2, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook3->Enabled = (Database->GetBookFieldInt(3, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook4->Enabled = (Database->GetBookFieldInt(4, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook5->Enabled = (Database->GetBookFieldInt(5, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook6->Enabled = (Database->GetBookFieldInt(6, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook7->Enabled = (Database->GetBookFieldInt(7, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook8->Enabled = (Database->GetBookFieldInt(8, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook9->Enabled = (Database->GetBookFieldInt(9, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
    btnBook10->Enabled = (Database->GetBookFieldInt(10, "Quantity") >= 1);
    if (Database->Error()) DisplayError();
}

```

```
void Delay(int secs)
{
    time_t start;
    time_t current;

    time(&start);
    do
    {
        time(&current);
    } while(difftime(current, start) < secs);
}

private: System::Void btnCancel_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    lblStatus->Visible = true;
    pnlDetail->Visible = false;
    pnlButtons->Visible = true;
    btnPurchase->Enabled = true;
    lblStatus->Text = gcnew String(GREETING);
}

private: System::Void btnPurchase_Click(System::Object^ sender,
                                       System::EventArgs^ e)
{
    String ^orig = gcnew String(lblISBN->Text->ToString());
    pin_ptr<const wchar_t> wch = PtrToStringChars(orig);

    // Convert to a char*
    size_t origsize = wcslen(wch) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    char nstring[newsize];
    wcstombs_s(&convertedChars, nstring, origsize, wch, _TRUNCATE);

    lblStatus->Visible = true;
    pnlDetail->Visible = false;
    pnlButtons->Visible = true;
    btnPurchase->Enabled = true;
    Database->VendBook(nstring);
    //
    // Simulate buying the book.
    //
    lblStatus->Text = "Please Insert your credit card.";
    this->Refresh();
    Delay(3);
    lblStatus->Text = "Thank you. Processing card number ending in 4-1234.";
    this->Refresh();
    Delay(3);
    lblStatus->Text = "Vending....";
    this->Refresh();
}
```



```
        Delay(5);
        this->Refresh();
        CheckAvailability();
        lblStatus->Text = gcnew String(GREETING);
    }

private: System::Void MainForm_Load(System::Object^ sender,
                                     System::EventArgs^ e)
{
    String ^imageName;
    String ^imagePath;

    Database->Initialize();
    if (Database->Error()) DisplayError();
    //
    //For each button, check to see if there are sufficient qty and load
    //the thumbnail for each.
    //
    imagePath = gcnew String(Database->GetSetting("ImagePath"));

    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(1, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook1->Image = btnBook1->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(2, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook2->Image = btnBook2->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(3, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook3->Image = btnBook3->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(4, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook4->Image = btnBook4->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(5, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook5->Image = btnBook5->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(6, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook6->Image = btnBook6->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(7, "Thumbnail"));
    if (Database->Error()) DisplayError();
    btnBook7->Image = btnBook7->Image->FromFile(imageName);
    imageName = imagePath +
        gcnew String(Database->GetBookFieldStr(8, "Thumbnail"));
```

```
if (Database->Error()) DisplayError();
btnBook8->Image = btnBook8->Image->FromFile(imageName);
imageName = imagePath +
    gcnew String(Database->GetBookFieldStr(9, "Thumbnail"));
if (Database->Error()) DisplayError();
btnBook9->Image = btnBook9->Image->FromFile(imageName);
imageName = imagePath +
    gcnew String(Database->GetBookFieldStr(10, "Thumbnail"));
if (Database->Error()) DisplayError();
btnBook10->Image = btnBook10->Image->FromFile(imageName);

CheckAvailability();
}
private: System::Void btnBook1_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(1);
}

private: System::Void btnBook2_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(2);
}

private: System::Void btnBook3_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(3);
}

private: System::Void btnBook4_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(4);
}

private: System::Void btnBook5_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(5);
}

private: System::Void btnBook6_Click(System::Object^ sender,
                                     System::EventArgs^ e)
{
    LoadDetails(6);
}

private: System::Void btnBook7_Click(System::Object^ sender,
```

```
        System::EventArgs^ e)
    {
        LoadDetails(7);
    }

private: System::Void btnBook8_Click(System::Object^ sender,
        System::EventArgs^ e)
    {
        LoadDetails(8);
    }

private: System::Void btnBook9_Click(System::Object^ sender,
        System::EventArgs^ e)
    {
        LoadDetails(9);
    }

private: System::Void btnBook10_Click(System::Object^ sender,
        System::EventArgs^ e)
    {
        LoadDetails(10);
    }
```



按钮和一些用来售货收钱的硬件装置。要说有什么不同的话，我的BVM机现在只接受信用卡付款，而一台真正的自动售货机还支持其他一些付款方式。

(2) 管理界面（管理窗体）

现在，BVM机已经有了一个简明易用的顾客界面。可那些图书数据应该如何去管理呢？如果出版公司需要补充或是改变BVM机销售的图书，它们该怎么做？一种办法是为BVM机再提供一个与顾客界面分开的管理界面。读者也可以另外创建一个嵌入式应用程序来做这些事情，还可以先在另一台机器上创建图书数据、再把它们复制到BVM机里去。我的选择是再创建一个简单的管理表单，如图6-8所示。

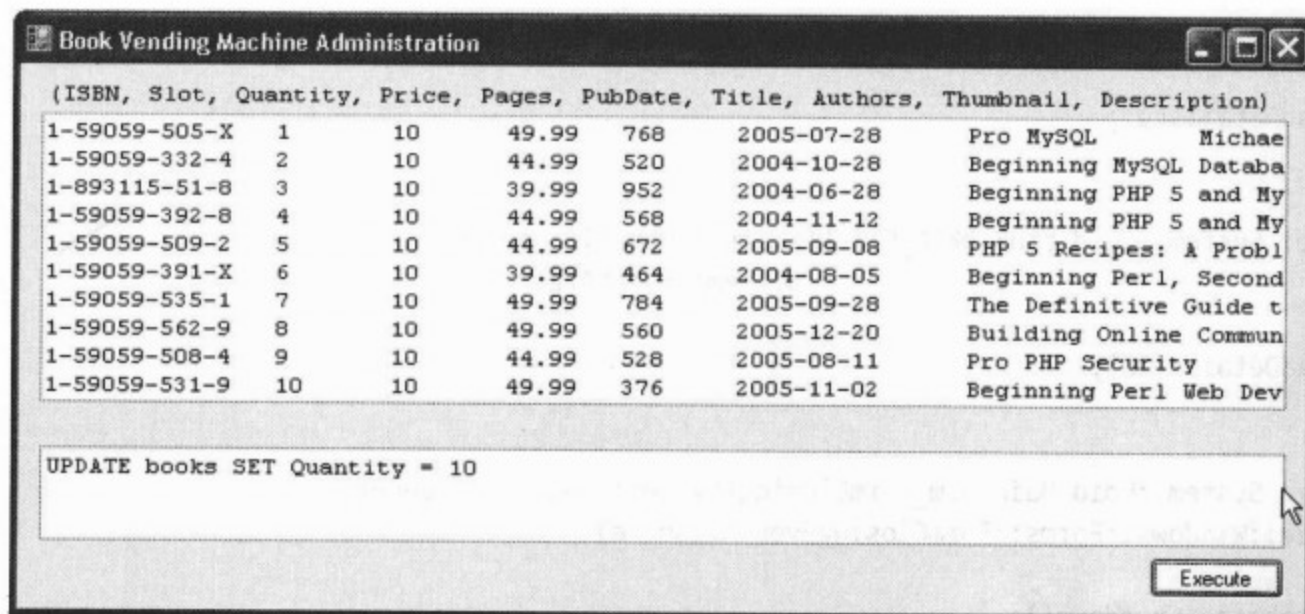


图6-8 示例管理窗体

类似于顾客界面的情况，我需要创建一个辅助函数。这个名为LoadList()的函数将把books表里的全部数据列成一份清单显示出来。这可以让出版公司的人员看到数据库里都包含着什么。

代码清单6-15给出了管理界面的源代码，我删节了许多由Visual Studio自动生成的代码（在代码里用“...”表示）。请注意，在这份清单的第2行就用#include "DBEngine.h"指令引入了DBEngine类的头文件，稍后几行又定义了一个DBEngine类型的变量。为了与用户操作界面有所区别，我把这个指针变量命名为AdminDatabase而不是Database——希望这不会令你困惑。代码里的...代表着被我删节的自动生成代码和注释。

代码清单6-15 管理窗体的源代码 (AdminForm.h)

```
#pragma once
#include "DBEngine.h"

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;
```



```

namespace BookVendingMachine {

    DBEngine *AdminDatabase = new DBEngine();

    ...
    #pragma endregion
    void LoadList()
    {
        int i = 0;
        int j = 0;
        String^ str;

        lstData->Items->Clear();
        AdminDatabase->StartQuery("SELECT ISBN, Slot, Quantity, Price, \" \
            \" Pages, PubDate, Title, Authors, Thumbnail, \" \
            \" Description FROM books");
        do
        {
            str = gcnew String("");
            for (i = 0; i < 10; i++)
            {
                if (i != 0)
                {
                    str = str + "\\t";
                }
                str = str + gcnew String(AdminDatabase->GetField(i));
            }
            lstData->Items->Add(str);
            j++;
        }while(AdminDatabase->GetNext());
    }

private: System::Void btnExecute_Click(System::Object^ sender,
                                       System::EventArgs^ e)
{
    String ^orig = gcnew String(txtQuery->Text->ToString());
    pin_ptr<const wchar_t> wch = PtrToStringChars(orig);

    // Convert to a char*
    size_t origsize = wcslen(wch) + 1;
    const size_t newsize = 100;
    size_t convertedChars = 0;
    char nstring[newsize];
    wcstombs_s(&convertedChars, nstring, origsize, wch, _TRUNCATE);
    AdminDatabase->RunQuery(nstring);
    LoadList();
}

private: System::Void Admin_Load(System::Object^ sender,
                                 System::EventArgs^ e)
{

```

```

        AdminDatabase->Initialize();
        LoadList();
    }

private: System::Void AdminForm_FormClosing(System::Object^ sender,
    System::Windows::Forms::FormClosingEventArgs^ e)
{
    AdminDatabase->Shutdown();
}
};
}

```

请注意，我把数据库引擎的初始化方法调用和关机方法调用分别放在了窗体加载事件和窗体关闭事件里。

我只给这个管理界面安排了一项工作——接受一条查询命令并在Execute按钮按下时执行，所以这份清单里的新东西只有btnExecute_Click()方法一个，其他代码都是现成的。btnExecute_Click()方法将把查询命令传递给数据库引擎去执行，但不检查任何结果。这是因为BVM机的管理界面只能用来修改数据库里的数据，不能选择数据。btnExecute_Click()方法里的最后一个调用是LoadList()辅助方法，在改好数据后会立刻看到一份新的图书清单。

6. 检测界面请求

既然BVM机有两个操作界面，怎样才能在它们之间进行切换呢？我选择的办法是用一个命令行参数来告诉代码应该进入哪个界面。我把这个开关实现在了BookVendingMachine.cpp源代码文件中的main()函数里。用来处理命令行参数的源代码很容易看懂。代码清单6-16完整地给出了这个嵌入式应用程序的main()函数的源代码。

代码清单6-16 BookVendingMachine主函数 (BookVendingMachine.cpp)

```

// BookVendingMachine.cpp : main project file.

#include "MainForm.h"
#include "AdminForm.h"

using namespace BookVendingMachine;

[STAThreadAttribute]
int main(array<System::String ^> ^args)
{
    // Enabling Windows XP visual effects before any controls are created
    Application::EnableVisualStyles();
    Application::SetCompatibleTextRenderingDefault(false);
    // Create the main window and run it
    if ((args->length == 1) && (args[0] == "-admin"))
    {
        Application::Run(gcnew AdminForm());
    }
    else
    {

```

```

        Application::Run(gcnew MainForm());
    }
    return 0;
}

```

如果想试试这个例子，可以按照前面描述的那些步骤，在自己的机器上把这个项目创建出来，也可以从本书的配套网站下载有关的源代码。建议仔细研究一下属于客户端（窗体）的源代码，这可以让你们熟悉和理解数据库引擎的用途和用法。准备好以后，就可以编译并运行这个例子了。

7. 编译和运行

编译这个示例程序非常简单，选择Build ► Build BookVendingMachine即可。如果你已经编译过libmysqld项目，这次将只编译这个示例程序；如果libmysqld库或其任意依赖的object文件因为某种原因过时了，Visual Studio会再次编译它们。

在编译工作结束后，就可以通过Visual Studio的调试菜单来运行这个程序了。另一个办法是打开一个命令窗口并从命令行来运行这个程序；从这个项目的根目录输入debug\BookVendingMachine命令。如果这是你第一次运行它，应该看到一条如下所示的出错消息。

```

This application has failed to start because LIBMYSQLD.dll was not found.
Re-installing the application may fix this problem.

```

这个错误的原因与出错消息里的第2句话没有任何关系，真正的根源是嵌入式库不在搜索路径上。如果你一直在与.NET或COM打交道，从没用过C语言库，可能从未见到过这种错误。与.NET或COM的情况不同，C语言库没有被注册到GAC（Global Assembly Cache，全局汇编缓存）或注册表里，所以在运行时找不到它们。你应该把这些库（DLL）与应用程序放在同一个子目录里，至少应该把它们放在某个搜索路径上。绝大多数程序员会把这些DLL复制到可执行文件所在的子目录里去。

为了解决这个问题，需要把libmysqld.dll文件从lib_debug子目录复制到bookvendingmachine.exe文件所在的子目录，或是把lib_debug子目录添加到搜索路径上。在解决这个问题之后，再次运行这个应用程序就可以看到如图6-3、图6-4和图6-5所示的效果了。

如果你想进入管理界面，需要使用-admin命令行开关来运行这个程序。如果你是从命令行运行这个例子的，可以输入如下所示的命令。

```
BookVendingMachine -admin
```

如果你是通过Visual Studio的调试器来运行这个程序的，需要在“项目属性”对话框里设置这个命令行开关。通过菜单命令Project ► Project Properties打开“项目属性”对话框，展开项目属性树的Debugging分支。可以添加任意个命令行参数，只要把它们输入到Command Arguments选项里就行了。这个选项在“项目属性”对话框里的位置如图6-9所示。

建议读者好好试试这个例子。即使你的操作系统不是Windows也没有关系，给这个例子数据库引擎类配上自己开发的操作界面就行了。这应该没什么问题，因为你已经见过一个如何利用经过抽象化的libmysqld系统调用去实现一个操作界面的例子了。如果你真的用嵌入式MySQL系统造出了一台独一无二的售货机，别忘了给我寄一张它的照片！

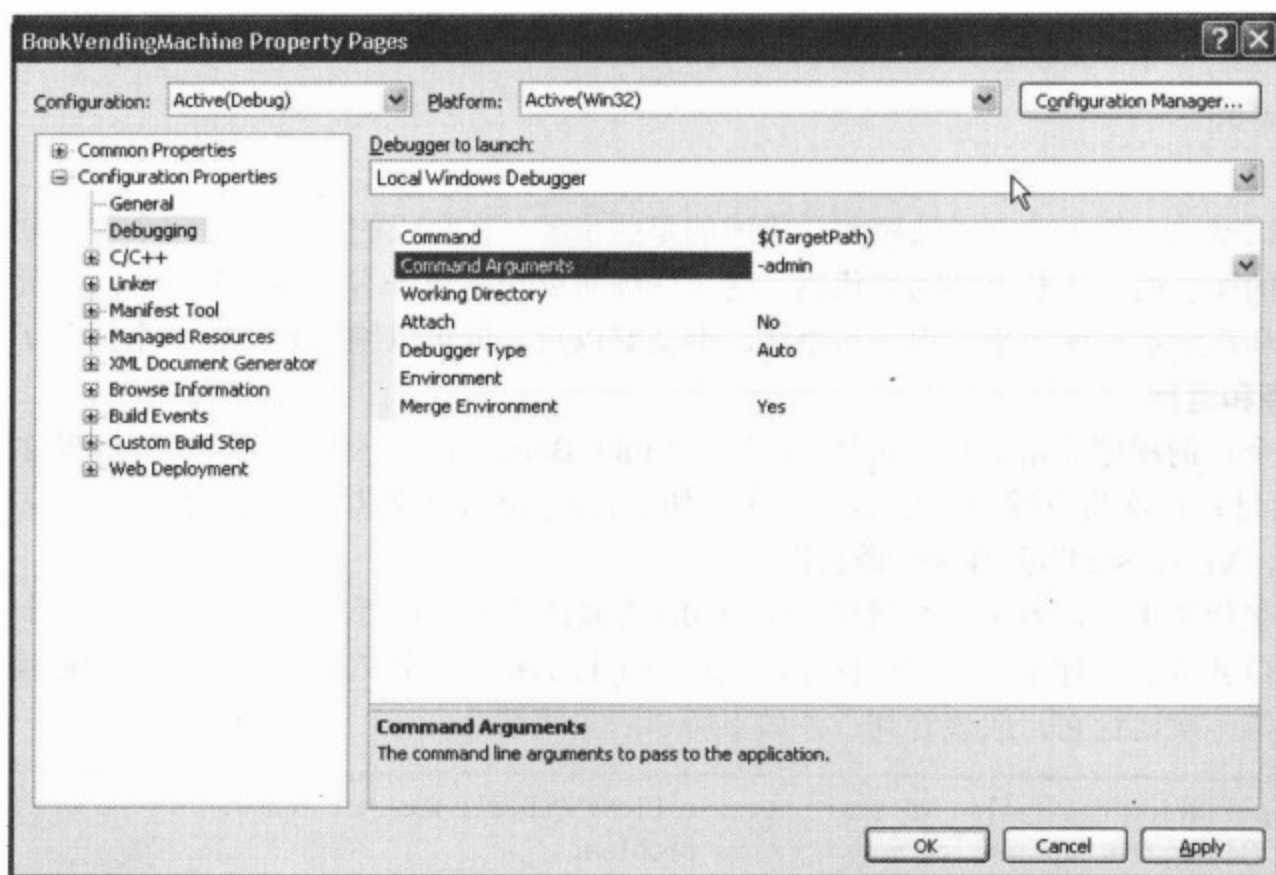


图6-9 在Visual Studio里设置命令行参数

6.5 小结

本章讲述了如何创建嵌入式MySQL应用程序。MySQL嵌入式库在MySQL家族里并不怎么引人注目，但系统集成商都知道，该库可以把健壮的数据管理功能集成到他们的企业级应用软件和产品里去。

本章最重要的内容应该是MySQL的嵌入式库C API了。希望本章的例子能够让你认识到嵌入式MySQL应用程序的强大。在编译源代码时候也许会遇到一些麻烦，但衷心希望你不会因此而抛开本书。如果想成为一名优秀的开源开发人员，就必须具备根据某个项目的具体情况去诊断和配置你的环境的能力。在遇到困难时不要灰心，学习的目的就是为了解决问题。

本章还介绍了几种在嵌入式应用程序里启用调试跟踪功能的概念。特别值得一提的是，通过让嵌入式库对外开放一个DEBUG方法来对MySQL服务器源代码进行修改，该库允许你把字符串添加到DEBUG踪迹输出里。本章还讲解了一些有趣的出错处理情况以及如何处理它们。最后，给出了一个封装了数据库访问功能的类，该类可在你自己的嵌入式应用程序里使用。

下一章将展示如何编写存储引擎。你会发现扩展MySQL来满足自己的需要并不像想象的那么困难，说不定你还会惊讶于它是如此的简单。单是嵌入式库就已经开启了无数的可能性，再加上存储引擎和MySQL函数，无怪乎人们把MySQL称为“世界上最受欢迎的开源数据库”。

插件式存储引擎是MySQL最重要的特性之一。能够根据存储数据的需要去调整关系数据库系统的物理存储机制，一直是数据库专家们的梦想，MySQL的插件式存储引擎把这一梦想变成了现实。利用MySQL的插件式存储引擎机制，数据库专家就可以对数据库系统的物理层进行调整和优化，就可以为数据库挑选一种能够让数据访问达到最优效果的存储方案。这是那些只支持一种存储机制^①的关系数据库系统无法比拟的巨大优势。

本章将介绍如何创建自己的存储引擎。这也是本书专门讨论如何修改和扩展MySQL系统的第一章。我将先解释一些插件式存储引擎的细节，然后介绍存储引擎的创建步骤，最后用一个示例存储引擎结束本章。如果你一直在寻找深入MySQL源代码的机会，它现在就摆在你的眼前。如果你对自己能否进行这样的修改没有信心，请反复阅读本章并按照本章的例子做练习，那么你最终会适应这个过程。

7.1 MySQL 插件式存储引擎概述

插件式存储引擎在MySQL服务器的体系结构里是一个软件层，它负责隔离MySQL服务器的物理数据层和逻辑层，并为服务器提供底层的输入/输出（I/O）操作。在一个有着层次化体系结构的系统里，各层次之间是通过一系列分工细致的标准化接口连接成一个整体的，那些接口的质量好坏决定着这个层次化体系结构的命运。层次化体系结构的最大优点是：只要接口不发生变化，对任何一个层次的修改都不会影响到与之相邻的其他层次。

MySQL AB公司从5.0版开始重新设计了MySQL服务器的体系结构。得到的最显而易见的成果之一就是插件式存储引擎机制出现在了5.1版的MySQL服务器里。插件式存储引擎使得系统集成厂商和程序员能够在数据需要经过特殊处理才能读写的环境里使用MySQL。不仅如此，插件式存储引擎机制还允许人们创建自己的存储引擎。

注解 在我写这本书的时候，MySQL的最新版本（5.1.9 β）还不能全面支持插入式机制。要想添加和识别一种新的存储引擎（通过`handlerton`，见后面的讨论），还需要对一些MySQL源文件进行必要的修改。按照MySQL AB公司的说法，MySQL系统的未来版本将不需要这些修改。

插件式存储引擎并不是唯一的解决方案，把数据转换为MySQL能够支持的某种格式也是一种选择。既然有两种选择，我们就不能不比较它们的成本。比如说，假设你们公司有一个使用了很多年的

^① 群集化索引和各种数据文件优化技术算不上是另外一种存储机制。

“老”程序，这个程序所使用的数据是你们公司的无价之宝，但因为种种原因无法复制，而你们公司的日常工作也离不开那个老程序。此时，与其把那些数据转换为一种新格式，创建一个存储引擎来读写老格式的数据往往更符合公司的利益。类似地，有些数据和/或它们的访问方法比较复杂，必须先对数据进行一些必要的预处理，才能保证数据读/写操作的效率。还有一种（也许是最重要的）情况是，插件式存储引擎可以把一些在过去被认为不适合存放在数据库系统里的数据存储到数据库里。比如说，你可以创建一个存储引擎来读取流式数据（如RSS等），或其他难以用传统的磁盘文件来保存的数据。总之，你现在可以通过自行创建一个存储引擎的办法来创建一个专用的关系数据库系统，无论你的要求是什么，MySQL都可以满足它。

你可以把MySQL服务器用作你的关系数据库处理引擎并让它直接读写你的“老”数据，而这一切只需要你提供一个可以直接插入MySQL服务器的专用存储引擎而已。这听起来似乎不是件简单的事情，但你们很快就会发现这其实没有多大困难。

最重要的体系结构元素是使用一系列单体对象去访问存储引擎（每种存储引擎对应一个对象）。这些单体对象是通过一个被称为handler的复杂结构来控制的（handler这个词源自singleton，请参见下面阴影部分的内容）。这些单体对象的基类是一个名为handler的类，从handler类衍生出来的handler对象提供了启用存储引擎所需要的接口和基本的连通性。

所有的存储引擎都是从handler基类衍生出来的，它就像是一名交通警察，来自逻辑层的数据访问方法和函数调用需要经过它才能到达硬件数据层，从硬件数据层返回的数据也需要经过它才能到达逻辑层。换句话说，handler和handler结构是存储引擎和服务器的一个中转站——更准确地说，是一个“黑箱”型中转站，只要你的存储引擎符合handler类的接口要求，你就可以把它插入到服务器里。所有的连接工作、身份验证工作、查询分析工作和查询优化工作，仍像往常那样由服务器来完成，来自不同存储介质的数据将由相应的存储引擎转换为一种统一的格式之后进入服务器，离开服务器的数据将由相应的存储引擎转换为某种特殊格式之后写入不同的存储介质。

MySQL AB公司在有关文档里对存储引擎的创建步骤做了相当细致的描述。在我编写这本书的时候，《MySQL参考手册》的第16章对存储引擎以及handler接口所支持和要求的所有函数都做了全面的解释。建议读者学完本章后好好读一读《MySQL参考手册》。《MySQL参考手册》就是用来参考的。

什么是singleton

在进行面向对象编程的时候，经常会遇到这样一种情况：你最多也只能为某个给定的类创建一个对象实例。这么做的理由之一是那个类保护着一组共享的操作或数据。比如说，假设你使用了一个manager类来检查用户是否有权访问某个特定的资源或数据。可以通过创建一个静态变量或全局变量来存储这个对象，以达到在整个应用程序里只允许有一个实例的目的，但使用全局实例和常数型结构或访问函数的做法，与面向对象的理念是背道而驰的。还有一个方法是，你可以创建一种特殊的对象来限制它只能有一个实例，并让应用程序里的其他对象共享该实例。这些特殊的、最多只能创建一次的对象就是所谓的singleton。（关于singleton的更多信息，请参阅www.codeproject.com/gen/design/singleton.asp上T. Kulathu Sarma写的文章*Creating Singleton Objects Using Visual*

C++。)下面是一些常见的创建singleton对象的办法:

- ☐ 静态变量
- ☐ 堆注册
- ☐ 实时类型信息 (runtime type information, RTTI)
- ☐ 自我注册
- ☐ 智能型singleton对象 (类似于智能型指针)

知道什么是singleton了吧? 在你的职业生涯里, 你一直都在创建singleton对象, 只是不知道它还有这么一个名字而已!

注解 插件式存储引擎并不是MySQL里唯一的插入式机制。MySQL还允许你使用插入式文本解析器和插入式用户定义函数。未来的MySQL版本还会包括插件式存储过程语言处理器。

7.1.1 基本过程

为MySQL服务器添加一种新存储引擎的过程可以划分为几个阶段。存储引擎不是一个只有几行或几十行代码的小程序, 开发一个如此庞大而复杂的东西, 必然需要有一个反复的过程: 先完成一小部分, 测试它, 再前进到下一个更复杂的部分。在接下来的教程里, 我将从最基本的函数开始逐步添加各项功能, 直到最终完成一个全功能的存储引擎为止。

前几个阶段是创建和添加基本的数据读写机制, 后几个阶段是添加索引功能和事务支持。根据你想让你自己的存储引擎具备哪些功能, 可能需要经历所有这些阶段。一个有实用价值的存储引擎至少应该支持在前4个阶段里定义的函数^①。下面几点对各阶段进行了描述。

(1) 生成引擎的方法存根 (Stubbing the engine) —— 整个过程的第一步是创建一个可以被插入服务器的基本的存储引擎。创建最基本的源代码文件, 为你的存储引擎从handler基类派生出一个子类, 把这个存储引擎本身插入到服务器的源代码里。

(2) 实现表 (文件) 操作 (Working with tables) —— 不能创建、打开、关闭和删除文件的存储引擎没有任何实际的意义。在这个阶段, 需要创建一些基本的文件处理例程并确保你的存储引擎能够正确地各种必要的文件操作。

(3) 实现数据读/写操作 (Reading and writing data) —— 为了完成最基本的存储引擎, 必须实现一些能够把数据读出和写入存储介质的读、写方法^②。在这个阶段, 需要添加一些能够以特定格式从存储介质上读出数据, 并把它们转换为MySQL内部数据格式的方法。类似地, 你可能还需要添加一些能够把MySQL内部数据格式转换为特定格式并写入存储介质的方法。

(4) 实现数据更新和删除操作 (Updating and deleting data) —— 如果你想让存储引擎更有用, 还需要实现一些可以对存储引擎里的数据进行修改的方法。在这个阶段, 你需要实现数据更新和

① 某些特殊的存储引擎可能根本不需要写数据。比如说, MySQL自备的BLACKHOLE存储引擎就没有实际实现任何写函数。它是数据宇宙中的黑洞!

② 这里要提醒大家一句: 没人规定数据必须保存在传统的数据存储介质上, 也没人规定只有保存在传统的数据存储介质上的东西才叫作数据。

删除操作。

(5) 添加索引功能 (Indexing the data) —— 一个好的存储引擎，还应该具备迅速完成随机读写操作和区间查询操作的能力。在这个阶段，你需要实现文件访问方法的复杂操作（该操作的复杂性仅次于需要你在下一阶段添加的事务支持功能）——建立索引。为了让这一阶段的工作变得容易一些，后面的例子封装了一个名为Spartan_index的索引类供大家使用和参考。

(6) 添加事务支持功能 (Adding transaction support) —— 整个过程的最后一步是为存储引擎添加事务支持功能。只有经过这一阶段，存储引擎才能真正成为一个可以在事务处理环境里使用的关系数据库存储机制。在文件访问方法里实现这些操作的复杂性是最大的。

在这一过程中，读者应该在每一个阶段都进行全面细致的测试和调试。在稍后的内容里，还将会看到我是如何调试一个存储引擎，以及如何为上述各个阶段编写测试程序并进行测试的。

7.1.2 需要用到的源文件

存储引擎的源文件通常有两个：一个代码文件（类文件）和一个头文件。这些文件被分别命名为ha_<engine name>.c（或*.cpp）和ha_<engine name>.h^①。比如说，归档存储引擎的文件被命名为ha_archive.cpp和ha_archive.h。存储引擎的源代码集中保存在MySQL源代码树根目录下的storage子目录里，在那个文件夹里可以找到各种各样的存储引擎的源代码文件。

7.1.3 其他辅助资源

《MySQL参考手册》提到了几个可以帮助你学习存储引擎的源代码文件。事实上，本章的大部分内容都来自我对那些资源的研究。MySQL AB公司提供了一个名为example的存储引擎，它是学习在阶段1创建存储引擎的最佳出发点。事实上，这个学习过程就是以此为起点的。

MySQL自带的归档引擎就是一个阶段3引擎，它提供了许多读、写数据的好例子。如果你想看到更多关于如何进行文件读、写和更新操作的例子，CSV引擎是个不错的选择。CSV引擎是一个阶段4存储引擎（CSV既可以读出和写入数据，也可以更新和删除数据）。CSV引擎是被最早实现出来的存储引擎之一，所以它与命名规则不同。它的源文件名为ha_tina.cc和ha_tina.h。最后，如果你想看看阶段5和阶段6存储引擎的例子，可以去研究MyISAM、InnoDB和BDB（Berkeley Database）存储引擎。

在开始创建自己的存储引擎之前，应该先把MySQL自带的这些存储引擎研究透彻，因为它们的源代码里有许多宝贵的建议和忠告，可以让读者对存储引擎应该如何工作有一个更深入细致的了解。如果你想扩展或模拟某个系统，先把该系统的内部工作情况搞清楚往往是最好的办法。

7.1.4 handlerton 类

我刚才讲过，handlerton类是所有存储引擎的标准接口。这个类的源代码可以在sql目录里的handler.cc和handler.h文件里查到。为了提供把存储引擎插入服务器和存储引擎完成其工作所需要的各种接口和功能，handlerton类还使用了许多其他的结构。

你们也许想知道，并发操作在这样一种机制里是如何得到保证的。答案是用另一个结构！每一种

^① MyIASM、InnoDB和BDB存储引擎还有其他的源文件，它们是最“古老”和最复杂的存储引擎。

存储引擎都有责任创建一个被所有线程里的处理函数的所有handler实例所共享的结构。显然，这意味着某些代码必须受到保护。还好你可以使用mutex（mutual exclusion，互斥保护机制）来保护你的代码；并且handlerton的源代码设计得非常合理，需要你采取这些保护措施的情况已经被降到了最低的限度。

handlerton结构是一个包含着许多数据项和方法的大型结构。在这个结构里定义的数据项都是很常见的数据类型，但这个结构里的方法都是用函数指针实现的。函数指针是功能最强大的结构化程序设计技术之一，它可以让程序员实现运行时多态。从而使函数指针可以让系统去执行另外一个函数（两个函数的调用接口必须一致）。正是因为运用了函数指针（和其他一些技术），handlerton结构才取得了如此巨大的成功。

代码清单7-1给出了handlerton结构的定义（有删节），表7-1对handlerton结构里比较重要的元素进行了说明。

注解 为了节省篇幅，省略了代码里的注释。为了简明扼要，又略去了一些不太重要的数据项。如果你想了解更多关于handlerton结构的信息，请阅读handler.h文件。

代码清单7-1 MySQL中的handlerton结构

```
typedef struct
{
    const int interface_version;
#define MYSQL_HANDLERTON_INTERFACE_VERSION 0x0001
    const char *name;
    SHOW_COMP_OPTION state;
    const char *comment;
    enum legacy_db_type db_type;
    bool (*init)();
    uint slot;
    uint savepoint_offset;
    int (*close_connection)(THD *thd);
    int (*savepoint_set)(THD *thd, void *sv);
    int (*savepoint_rollback)(THD *thd, void *sv);
    int (*savepoint_release)(THD *thd, void *sv);
    int (*commit)(THD *thd, bool all);
    int (*rollback)(THD *thd, bool all);
    int (*prepare)(THD *thd, bool all);
    int (*recover)(XID *xid_list, uint len);
    int (*commit_by_xid)(XID *xid);
    int (*rollback_by_xid)(XID *xid);
    void *(*create_cursor_read_view)();
    void (*set_cursor_read_view)(void *);
    void (*close_cursor_read_view)(void *);
    handler *(*create)(TABLE_SHARE *table);
    void (*drop_database)(char* path);
    int (*panic)(enum ha_panic_function flag);
    int (*start_consistent_snapshot)(THD *thd);
    bool (*flush_logs)();
}
```

```

bool (*show_status)(THD *thd, stat_print_fn *print, enum ha_stat_type stat);
uint (*partition_flags)();
uint (*alter_table_flags)(uint flags);
int (*alter_tablespace)(THD *thd, st_alter_tablespace *ts_info);
int (*fill_files_table)(THD *thd,
                        struct st_table_list *tables,
                        class Item *cond);

uint32 flags; /* global handler flags */
int (*binlog_func)(THD *thd, enum_binlog_func fn, void *arg);
void (*binlog_log_query)(THD *thd, enum_binlog_command binlog_command,
                        const char *query, uint query_length,
                        const char *db, const char *table_name);
int (*release_temporary_latches)(THD *thd);
} handlerton;

```

注解 “类型” 栏里值的含义如下: const=常数, var=变量, enum=枚举集合, fptr=函数指针。

表7-1 handlerton结构

元 素	说 明
const char *name	存储引擎的名字; 服务器将使用这个名字来识别各存储引擎; 可以用“SHOW STORAGE ENGINES;”命令查看
SHOW_COMP_OPTION state	检查存储引擎是否可用
const char *comment	一条对存储引擎进行描述的注释; 可以用SHOW命令查看
enum legacy_db_type db_type	保存在*.frm文件里的一个枚举值, 用来表明这个文件是用来创建哪一种存储引擎的。这个值还被用来确定与某给定表相关联的存储引擎(handler类)是哪一个
bool (*init)()	对存储引擎(handler类)进行初始化。为存储引擎分配内存
uint slot	这个handlerton对象在存储引擎表里的位置编号
uint savepoint_offset	为这个存储引擎创建保存点(savepoint)所需要的内存量
int (*close_connection) (...)	关闭当前连接
int (*savepoint_set) (...)	把保存点设置为savepoint_offset元素指定的保存点偏移值
int (*savepoint_rollback) (...)	回到一个保存点(撤销该保存点之后执行操作)
int (*savepoint_release) (...)	释放(忽略)一个保存点
int (*commit) (...)	提交一个事务
int (*rollback) (...)	撤销一个事务
int (*prepare) (...)	准备提交一个事务
int (*recover) (...)	撤销一些尚未提交的事务
int (*commit_by_xid) (...)	根据给定的事务ID提交一个事务
int (*rollback_by_xid) (...)	根据给定的事务ID撤销一个事务
void *(*create_cursor_read_view) ()	创建一个游标(读写指针)
void (*set_cursor_read_view) (void *)	切换到指定的游标视图
void (*close_cursor_read_view) (void *)	关闭指定的游标视图
handler *(*create) (TABLE_SHARE *table)	为这个存储引擎创建一个handler实例
int (*panic) (enum ha_panic_function flag)	这个方法会在服务器关机和崩溃时被调用

(续)

元 素	说 明
<code>int (*start_consistent_snapshot) (...)</code>	开始一次连续读操作 (支持并发处理)
<code>bool (*flush_logs) ()</code>	把缓存在内存里的日志信息写入磁盘
<code>bool (*show_status) (...)</code>	返回存储引擎的状态信息
<code>uint (*partition_flags) ()</code>	返回启用/禁用各项监控功能的开关标志字
<code>uint (*alter_table_flags) (...)</code>	返回“ALTER TABLE”命令的功能开关标志字
<code>int (*alter_tablespace) (...)</code>	返回“ALTER TABLESPACE”命令的功能开关标志字
<code>int (*fill_files_table) (...)</code>	群集服务器机制将使用这个方法来填充表 (请参见NDB存储引擎的文档)
<code>uint32 flags</code>	一个用来表明这个存储引擎都支持哪些功能的标志字
<code>int (*binlog_func) (...)</code>	用来反调用二进制日志函数的方法
<code>void (*binlog_log_query) (...)</code>	用来查询二进制日志的方法
<code>int (*release_temporary_latches) (...)</code>	一个InnoDB专用方法 (请参见InnoDB存储引擎的文档)

7.1.5 handler 类

handler类是理解插件式存储引擎接口的另一个关键。handler类派生自Sql_alloc类,这意味着它通过继承获得了所有的内存分配例程。handler类对存储引擎的接口部分做出了定义,提供了一整套通过handler_ton结构与服务器打交道的接口方法。handler_ton和handler实例共同构成了存储引擎体系结构的抽象层。如图7-1所示,可以从这两个类派生出新的存储引擎,而handler_ton结构是handler类和新存储引擎之间的一个接口。

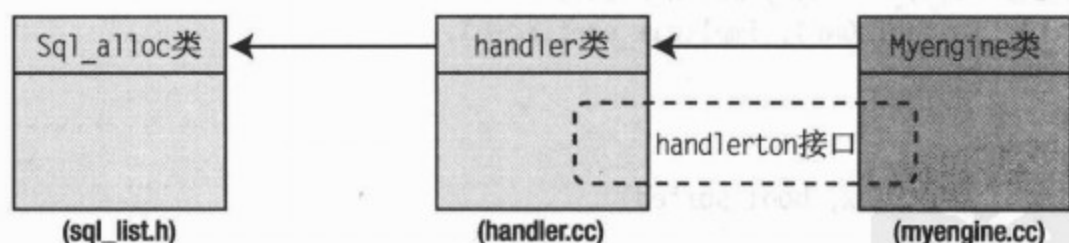


图7-1 插件式存储引擎类型的派生关系

对handler类的详细分析超出了本书的讨论范围。我将在本章后面的内容里,在实现一个示例存储引擎的过程中把handler类最重要和最常用的方法介绍给大家。

为了让大家对handler类有一个基本的了解,我从handler类的定义里节选了一些内容,列在了代码清单7-2里。快速浏览一下这份清单就会发现,handler类包含着许多方法,它们可以用来完成各种各样的操作 (如创建表、删除表、改变表的结构等),还有一些方法是用来对字段和索引进行处理的,甚至还有用于崩溃保护、恢复和备份的方法。

handler类提供的方法覆盖了一个存储引擎可能会遇到的每一种情况,但绝大多数存储引擎并不会把所有的方法都实现出来。如果你想利用handler类提供的某些高级功能实现一个存储引擎,应该多花些时间去研究《MySQL参考手册》里对handler类的详细介绍。熟悉了创建存储引擎的过程之后,可以运用这个参考手册里学到的知识把存储引擎的功能成熟度提高到一个新的水平。

代码清单7-2 handler类的定义（节选）

```

class handler :public Sql_alloc
{
...
    const handlerton *ht;           /* storage engine of this handler */
    byte *ref;                      /* Pointer to current row */
    byte *dupp_ref;                 /* Pointer to dupp row */
    ulonglong data_file_length;     /* Length of data file */
    ulonglong max_data_file_length; /* Length of data file */
    ulonglong index_file_length;
    ulonglong max_index_file_length;
    ulonglong delete_length;        /* Free bytes */
    ulonglong auto_increment_value;
    ha_rows records;               /* Records in table */
    ha_rows deleted;               /* Deleted records */
    ulong mean_rec_length;         /* physical relength */
    time_t create_time;            /* When table was created */
    time_t check_time;
    time_t update_time;
...
    handler(const handlerton *ht_arg, TABLE_SHARE *share_arg)
        :table_share(share_arg), ht(ht_arg),
        ref(0), data_file_length(0), max_data_file_length(0), index_file_length(0),
        delete_length(0), auto_increment_value(0),
        records(0), deleted(0), mean_rec_length(0),
        create_time(0), check_time(0), update_time(0),
        key_used_on_scan(MAX_KEY), active_index(MAX_KEY),
        ref_length(sizeof(my_off_t)), block_size(0),
        ft_handler(0), inited(NONE), implicit_emptyed(0),
        pushed_cond(NULL)
    {}
...
    int ha_index_init(uint idx, bool sorted)
...
    int ha_index_end()
...
    int ha_rnd_init(bool scan)
...
    int ha_rnd_end()
...
    int ha_reset()
...
...
    virtual int exec_bulk_update(uint *dup_key_found)
...
    virtual void end_bulk_update() { return; }
...
    virtual int end_bulk_delete()
...

```



```

virtual int index_read(byte * buf, const byte * key,
    uint key_len, enum ha_rkey_function find_flag)
...
virtual int index_read_idx(byte * buf, uint index, const byte * key,
    uint key_len, enum ha_rkey_function find_flag);
virtual int index_next(byte * buf)
    { return HA_ERR_WRONG_COMMAND; }
virtual int index_prev(byte * buf)
    { return HA_ERR_WRONG_COMMAND; }
virtual int index_first(byte * buf)
    { return HA_ERR_WRONG_COMMAND; }
virtual int index_last(byte * buf)
    { return HA_ERR_WRONG_COMMAND; }
virtual int index_next_same(byte *buf, const byte *key, uint keylen);
virtual int index_read_last(byte * buf, const byte * key, uint key_len)
...
virtual int read_multi_range_first(KEY_MULTI_RANGE **found_range_p,
    KEY_MULTI_RANGE *ranges, uint range_count,
    bool sorted, HANDLER_BUFFER *buffer);
virtual int read_multi_range_next(KEY_MULTI_RANGE **found_range_p);
virtual int read_range_first(const key_range *start_key,
    const key_range *end_key,
    bool eq_range, bool sorted);
virtual int read_range_next();
int compare_key(key_range *range);
virtual int ft_init() { return HA_ERR_WRONG_COMMAND; }
void ft_end() { ft_handler=NULL; }
virtual FT_INFO *ft_init_ext(uint flags, uint inx,String *key)
    { return NULL; }
virtual int ft_read(byte *buf) { return HA_ERR_WRONG_COMMAND; }
virtual int rnd_next(byte *buf)=0;
virtual int rnd_pos(byte * buf, byte *pos)=0;
virtual int read_first_row(byte *buf, uint primary_key);
...
virtual int restart_rnd_next(byte *buf, byte *pos)
    { return HA_ERR_WRONG_COMMAND; }
virtual int rnd_same(byte *buf, uint inx)
    { return HA_ERR_WRONG_COMMAND; }
virtual ha_rows records_in_range(uint inx, key_range *min_key,
    key_range *max_key)
    { return (ha_rows) 10; }
virtual void position(const byte *record)=0;
virtual void info(uint)=0; // see my_base.h for full description
virtual void get_dynamic_partition_info(PARTITION_INFO *stat_info,
    uint part_id);
virtual int extra(enum ha_extra_function operation)
    { return 0; }
virtual int extra_opt(enum ha_extra_function operation, ulong cache_size)
    { return extra(operation); }
...

```

```

    virtual int delete_all_rows()
...
    virtual ulonglong get_auto_increment();
    virtual void restore_auto_increment();
...
    virtual int reset_auto_increment(ulonglong value)
...
    virtual void update_create_info(HA_CREATE_INFO *create_info) {}
...
    int ha_repair(THD* thd, HA_CHECK_OPT* check_opt);
...
    virtual bool check_and_repair(THD *thd) { return HA_ERR_WRONG_COMMAND; }
    virtual int dump(THD* thd, int fd = -1) { return HA_ERR_WRONG_COMMAND; }
    virtual int disable_indexes(uint mode) { return HA_ERR_WRONG_COMMAND; }
    virtual int enable_indexes(uint mode) { return HA_ERR_WRONG_COMMAND; }
    virtual int indexes_are_disabled(void) {return 0;}
    virtual void start_bulk_insert(ha_rows rows) {}
    virtual int end_bulk_insert() {return 0; }
    virtual int discard_or_import_tablespace(my_bool discard)
...
    virtual uint referenced_by_foreign_key() { return 0;}
    virtual void init_table_handle_for_HANDLER()
...
    virtual void free_foreign_key_create_info(char* str) {}
...
    virtual const char *table_type() const =0;
    virtual const char **bas_ext() const =0;
    virtual ulong table_flags(void) const =0;
...
    virtual uint max_supported_record_length() const { return HA_MAX_REC_LENGTH; }
    virtual uint max_supported_keys() const { return 0; }
    virtual uint max_supported_key_parts() const { return MAX_REF_PARTS; }
    virtual uint max_supported_key_length() const { return MAX_KEY_LENGTH; }
    virtual uint max_supported_key_part_length() const { return 255; }
    virtual uint min_record_length(uint options) const { return 1; }
...
    virtual bool is_crashed() const { return 0; }
...
    virtual int rename_table(const char *from, const char *to);
    virtual int delete_table(const char *name);
    virtual void drop_table(const char *name);

    virtual int create(const char *name, TABLE *form, HA_CREATE_INFO *info)=0;
...
    virtual int external_lock(THD *thd __attribute__((unused)),
                             int lock_type __attribute__((unused)))
...
    virtual int write_row(byte *buf __attribute__((unused)))
...
    virtual int update_row(const byte *old_data __attribute__((unused)),

```

```

        byte *new_data __attribute__((unused)))
...
    virtual int delete_row(const byte *buf __attribute__((unused)))
...
};

```

7.1.6 对 MySQL 存储引擎的简要分析

想了解handler类是如何工作的？那就在它工作时去观察好了。在开始创建一个新的存储引擎之前，有必要先去看看一个真正的存储引擎是如何工作的。首先，请在调试模式下编译你的服务器——如果你还没有这样做过的话。然后，启动你的服务器和调试器并按照第5章给出的步骤把调试工具关联到正在运行的服务器。

先看一个简单的存储引擎是如何工作的。具体地说，我将使用归档存储引擎作为例子。在启动调试器和服务器之后，请打开ha_archive.cc文件并在以下几个方法的第一条可执行语句上设置一个断点。

```

❑ int ha_archive::create (...)
❑ static ARCHIVE_SHARE * ha_archive::get_share (...)
❑ int ha_archive::write_row (...)
❑ int ha_archive::rnd_next(...)

```

把断点设置好以后，启动MySQL命令行客户端程序，进入test数据库，输入下面这条命令。

```
CREATE TABLE testarc (a int, b varchar(20), c int) ENGINE=ARCHIVE;
```

你们应该立刻看到调试器停在了create()方法里。这个方法是创建表的地方。事实上，它是最早执行的操作之一。为创建这个文件，这里调用了my_create()方法。请注意，在这个方法里，字段迭代器遍历表里的所有字段。这很重要，因为它表明所有字段已被创建。它们被保存在数据文件夹里的testarc.frm文件中。该代码还会检查是不是有设置了AUTO_INCREMENT_FLAG标志的字段（在这个方法的开头部分）；如果真的找到了一个这样的字段，代码将报告出错并退出执行，这是因为归档存储引擎不支持自动递增字段。还可以看到这个方法创建了一个元文件并检查了数据压缩例程是否在正常工作。

请以单步方式调试这段代码并观察迭代器的工作情况。随时都可以让程序继续执行，但如果你真的很好奇，不妨以单步方式一直执行到这个函数调用返回为止。

现在，我们来看看在插入数据时会发生什么事情。回到MySQL客户端程序并输入下面这条命令。

```
INSERT INTO testarc VALUES (10, "test", -1);
```

这一次，代码停在了get_share()方法里。这个方法负责为归档处理函数的所有实例创建一个共享结构。如果你以单步方式观察这个方法的工作情况，就可以看到代码是在什么地方设置全局变量以及进行其他初始化处理的。如果没有其他让你感兴趣的东西，请让调试器继续执行。

代码下一次停止执行的地方是在write_row()方法里。这个方法将把被传递到buf参数里的数据写到磁盘上。这个记录缓冲区（byte *buf）是MySQL用来在其内部传递行的机制。它是一个二进制缓冲区，其内容是行的数据和其他一些元数据——这些东西在MySQL的随机文档里被称为“内部格式”。如果以单步调试的方式运行这段代码，将会看到这个引擎设置了一些统计数据，又进行了一些出错检查，最后在这个方法的末尾调用real_write_row()方法把数据写到了磁盘。请以单步方式进入

real_write_row()方法。

在real_write_row()方法里，你可以看到另外一个字段迭代器。这个迭代器将遍历各个BLOB (binary large object, 二进制大对象) 字段并用数据压缩的方法把其中的数据写入磁盘。如果你需要支持BLOB字段，这是一个学习应该如何去做的绝佳例子——只要把你的底层I/O调用替换为某个数据压缩方法就行了。如果没有其他让你感兴趣的东西，请让调试器继续执行。然后，回到MySQL客户端程序并输入下面这条命令：

```
SELECT * FROM testarc;
```

代码下一次停止执行的地方是在rnd_next()方法里。处理函数就是在这里读取数据文件并把数据返回到记录缓冲区 (byte *buf) 里的。我们又一次看到，有关代码设置了一些统计数据，进行了一些出错检查，然后调用get_row()方法把数据读入内存。对此段代码略微执行一下单步调试，然后让调试器继续执行。

让人惊讶的是，代码再次停在了rnd_next()方法里。这是因为rnd_next()方法是全表扫描过程中的一系列调用之一。这个方法不仅负责从磁盘读入数据，还负责检查是否已经到达文件尾。于是，在你正在调试的例子里，这个方法被调用了两次。第一次是检索第一个行，第二次是检查是否到达文件尾（你只插入了一个行）。下面的清单是对你们正在调试的例子进行一次全表扫描的典型调用序列。

```
ha_spartan::info
ha_spartan::rnd_init
ha_spartan::extra
ha_spartan::rnd_next
ha_spartan::rnd_next
ha_spartan::extra

+-----+-----+-----+
| a     | b     | c     |
+-----+-----+-----+
| 10    | test  | -1    |
+-----+-----+-----+
1 row in set (26.25 sec)
```

注解 这个查询命令返回的时间是由服务器记录下来的实际经过的时间，不是实际执行所花费的时间。因此，这个时间包含了调试这些代码所花费的时间。

你可以在自己感兴趣的方法上随意设置断点。还可以花些时间去读读这个存储引擎里的注释，它们提供的宝贵线索可以让你们进一步了解某些处理方法的用途。

7.2 Spartan 存储引擎

我为这个关于存储引擎的教程准备的例子是一个基本的存储引擎，一个普通的存储引擎应该具备的功能它都具备。这包括读、写数据以及建立和使用索引。换句话说，它是一个阶段5引擎。我把这个示例存储引擎称为Spartan存储引擎，它只实现了一个有实用价值的存储引擎必须具备的最基本的功能。

我将从MySQL示例存储引擎 (ha_example) 开始，带领你一步步完成Spartan存储引擎的构建工作。在这个过程中，我会随时把关于其他存储引擎的知识和信息介绍给你。你也许会发现一些自认为应该

改进的地方（它也确实有几个需要改进的地方），但在把Spartan引擎成功地实现到阶段5的水平之前，应该先把改进它的念头放在一边。

我们就从分析Spartan存储引擎的支持类文件开始这次旅程吧。

7.2.1 底层 I/O 类

一个存储引擎的价值体现在它可以通过某种特殊的机制去读、写数据并为用户带来一些独特的好处。这意味着任意两个存储引擎本质上不会支持同样的功能。在MySQL自带的存储引擎当中，有几个把底层的I/O函数也嵌在了自己的源文件里。

底层I/O功能的实现不外乎两种，一种是存放在其他源文件里的C函数；另一种是定义在类头文件和类源文件里的C++类。具体到Spartan存储引擎，我决定采用后一种做法。我创建了一个数据文件类和一个索引文件类。因为本章和Spartan存储引擎项目的重点是如何创建一个存储引擎，所以我没有对这两个类进行性能方面的优化。尽管如此，它们毕竟是我为了创建一个可以工作的存储引擎而准备的工具，在创建自己的存储引擎时需要做的事情几乎都可以在它们那里找到样板或演示。

因为篇幅的限制，本节只能对这两个类做概括性的描述，但你可以通过阅读它们的代码去了解它们的工作细节。虽然这两个底层I/O类都很初级，都有不少值得改进的地方，但我相信使用这两个类会得到很多好处。甚至可以把它们当作自己的存储引擎I/O机制的基础。

1. Spartan_data类

Spartan_data类是Spartan存储引擎最主要的底层I/O类。这个类的主要工作是为Spartan存储引擎封装数据。代码清单7-3完整地给出了这个类的头文件代码。正如你们将在这个头文件里看到的那样，这个类的方法都比较简单。我只实现了基本的打开、关闭、读和写操作。

代码清单7-3 Spartan_data类的头文件 (Spartan_data.h)

```
/*
    spartan_data.h

    This header defines a simple data file class for reading raw data to and
    from disk. The data written is in byte format so it can be anything you
    want it to be. The write_row and read_row accept the length of the data
    item to be read.
*/
#pragma once
#pragma unmanaged
#include "my_global.h"
#include "my_sys.h"
class Spartan_data
{
public:
    Spartan_data(void);
    ~Spartan_data(void);
    int create_table(char *path);
    int open_table(char *path);
    long long write_row(byte *buf, int length);
    long long update_row(byte *old_rec, byte *new_rec,
```

```

        int length, long long position);
int read_row(byte *buf, int length, long long position);
int delete_row(byte *old_rec, int length, long long position);
int close_table();
long long cur_position();
int records();
int del_records();
int trunc_table();
int row_size(int length);
private:
    File data_file;
    int header_size;
    int record_header_size;
    bool crashed;
    int number_records;
    int number_del_records;
    int read_header();
    int write_header();
};

```

代码清单7-4完整地给出了Spartan存储引擎的数据类（Spartan_data类）的源代码。请注意，我在代码里加入了一些DEBUG调用，它们可以在我使用--with-debug开关调试这个系统时，确保我的源代码可以写出一份踪迹文件。还请注意，这段代码在读、写有关数据的时候，使用的都是MySQL AB公司提供的my_xxx()形式的工具方法，它们在跨平台使用时一般不会出现兼容问题。

代码清单7-4 Spartan_data类的源代码（Spartan_data.cpp）

```

/*
Spartan_data.cpp

This class implements a simple data file reader/writer. It
is designed to allow the caller to specify the size of the
data to read or write. This allows for variable length records
and the inclusion of extra fields (like BLOBs). The data is
stored in an uncompressed, unoptimized fashion.
*/
#include "spartan_data.h"
#include <my_dir.h>
#include <string.h>

Spartan_data::Spartan_data(void)
{
    data_file = -1;
    number_records = -1;
    number_del_records = -1;
    header_size = sizeof(bool) + sizeof(int) + sizeof(int);
    record_header_size = sizeof(byte) + sizeof(int);
}

Spartan_data::~Spartan_data(void)

```

```
{
}

/* create the data file */
int Spartan_data::create_table(char *path)
{
    DEBUG_ENTER("Spartan_data::create_table");
    open_table(path);
    number_records = 0;
    number_del_records = 0;
    crashed = false;
    write_header();
    DEBUG_RETURN(0);
}

/* open table at location "path" = path + filename */
int Spartan_data::open_table(char *path)
{
    DEBUG_ENTER("Spartan_data::open_table");
    /*
     * Open the file with read/write mode,
     * create the file if not found,
     * treat file as binary, and use default flags.
     */
    data_file = my_open(path, O_RDWR | O_CREAT | O_BINARY | O_SHARE, MYF(0));
    if(data_file == -1)
        DEBUG_RETURN(errno);
    read_header();
    DEBUG_RETURN(0);
}

/* write a row of length bytes to file and return position */
long long Spartan_data::write_row(byte *buf, int length)
{
    long long pos;
    int i;
    int len;
    byte deleted = 0;

    DEBUG_ENTER("Spartan_data::write_row");
    /*
     * Write the deleted status byte and the length of the record.
     * Note: my_write() returns the bytes written or -1 on error
     */
    pos = my_seek(data_file, 0L, MY_SEEK_END, MYF(0));
    /*
     * Note: my_malloc takes a size of memory to be allocated,
     * MySQL flags (set to zero fill and with extra error checking).
     * Returns number of bytes allocated -- <= 0 indicates an error.
     */
    i = my_write(data_file, &deleted, sizeof(byte), MYF(0));
```

```

memcpy(&len, &length, sizeof(int));
i = my_write(data_file, (byte *)&len, sizeof(int), MYF(0));
/*
    Write the row data to the file. Return new file pointer or
    return -1 if error from my_write().
*/
i = my_write(data_file, buf, length, MYF(0));
if (i == -1)
    pos = i;
else
    number_records++;
DEBUG_RETURN(pos);
}

/* update a record in place */
long long Spartan_data::update_row(byte *old_rec, byte *new_rec,
                                   int length, long long position)
{
    long long pos;
    long long cur_pos;
    byte *cmp_rec;
    int len;
    byte deleted = 0;
    int i = -1;
    DEBUG_ENTER("Spartan_data::update_row");
    if (position == 0)
        position = header_size; //move past header
    pos = position;
    /*
        If position unknown, scan for the record by reading a row
        at a time until found.
    */
    if (position == -1) //don't know where it is...scan for it
    {
        cmp_rec = (byte *)my_malloc(length, MYF(MY_ZEROFILL | MY_WME));
        pos = 0;
        /*
            Note: my_seek() returns pos if no errors or -1 if error.
        */
        cur_pos = my_seek(data_file, header_size, MY_SEEK_SET, MYF(0));
        /*
            Note: read_row() returns current file pointer if no error or
            -1 if error.
        */
        while ((cur_pos != -1) && (pos != -1))
        {
            pos = read_row(cmp_rec, length, cur_pos);
            if (memcmp(old_rec, cmp_rec, length) == 0)
            {

```



```

        pos = cur_pos;        //found it!
        cur_pos = -1;        //stop loop gracefully
    }
    else if (pos != -1) //move ahead to next rec
        cur_pos = cur_pos + length + record_header_size;
    }
    my_free((gptr)cmp_rec, MYF(0));
}
/*
If position found or provided, write the row.
*/
if (pos != -1)
{
    /*
    Write the deleted byte, the length of the row, and the data
    at the current file pointer.
    Note: my_write() returns the bytes written or -1 on error
    */
    my_seek(data_file, pos, MY_SEEK_SET, MYF(0));
    i = my_write(data_file, &deleted, sizeof(byte), MYF(0));
    memcpy(&len, &length, sizeof(int));
    i = my_write(data_file, (byte *)&len, sizeof(int), MYF(0));
    pos = i;
    i = my_write(data_file, new_rec, length, MYF(0));
}
DEBUG_RETURN(pos);
}

/* delete a record in place */
int Spartan_data::delete_row(byte *old_rec, int length,
                             long long position)
{
    int i = -1;
    long long pos;
    long long cur_pos;
    byte *cmp_rec;
    byte deleted = 1;

    DEBUG_ENTER("Spartan_data::delete_row");
    if (position == 0)
        position = header_size; //move past header
    pos = position;
    /*
    If position unknown, scan for the record by reading a row
    at a time until found.
    */
    if (position == -1) //don't know where it is...scan for it
    {
        cmp_rec = (byte *)my_malloc(length, MYF(MY_ZEROFILL | MY_WME));
        pos = 0;
    }

```

```

/*
    Note: my_seek() returns pos if no errors or -1 if error.
*/
cur_pos = my_seek(data_file, header_size, MY_SEEK_SET, MYF(0));
/*
    Note: read_row() returns current file pointer if no error or
    -1 if error.
*/
while ((cur_pos != -1) && (pos != -1))
{
    pos = read_row(cmp_rec, length, cur_pos);
    if (memcmp(old_rec, cmp_rec, length) == 0)
    {
        number_records--;
        number_del_records++;
        pos = cur_pos;
        cur_pos = -1;
    }
    else if (pos != -1) //move ahead to next rec
        cur_pos = cur_pos + length + record_header_size;
}
my_free((gptr)cmp_rec, MYF(0));
}
/*
    If position found or provided, write the row.
*/
if (pos != -1) //mark as deleted
{
    /*
        Write the deleted byte set to 1 which marks row as deleted
        at the current file pointer.
        Note: my_write() returns the bytes written or -1 on error
    */
    pos = my_seek(data_file, pos, MY_SEEK_SET, MYF(0));
    i = my_write(data_file, &deleted, sizeof(byte), MYF(0));
    i = (i > 1) ? 0 : i;
}
DEBUG_RETURN(i);
}

/* read a row of length bytes from file at position */
int Spartan_data::read_row(byte *buf, int length, long long position)
{
    int i;
    int rec_len;
    long long pos;
    byte deleted = 2;

    DEBUG_ENTER("Spartan_data::read_row");
    if (position <= 0)

```

```

    position = header_size; //move past header
    pos = my_seek(data_file, position, MY_SEEK_SET, MYF(0));
    /*
    If my_seek found the position, read the deleted byte.
    Note: my_read() returns bytes read or -1 on error
    */
    if (pos != -1L)
    {
        i = my_read(data_file, &deleted, sizeof(byte), MYF(0));
        /*
        If not deleted (deleted == 0), read the record length then
        read the row.
        */
        if (deleted == 0) /* 0 = not deleted, 1 = deleted */
        {
            i = my_read(data_file, (byte *)&rec_len, sizeof(int), MYF(0));
            i = my_read(data_file, buf,
                (length < rec_len) ? length : rec_len, MYF(0));
        }
        else if (i == 0)
            DEBUG_RETURN(-1);
        else
            DEBUG_RETURN(read_row(buf, length, cur_position() +
                length + (record_header_size - sizeof(byte))));
    }
    else
        DEBUG_RETURN(-1);
    DEBUG_RETURN(0);
}

/* close file */
int Spartan_data::close_table()
{
    DEBUG_ENTER("Spartan_data::close_table");
    if (data_file != -1)
    {
        my_close(data_file, MYF(0));
        data_file = -1;
    }
    DEBUG_RETURN(0);
}

/* return number of records */
int Spartan_data::records()
{
    DEBUG_ENTER("Spartan_data::num_records");
    DEBUG_RETURN(number_records);
}

/* return number of deleted records */
int Spartan_data::del_records()

```

```
{
    DEBUG_ENTER("Spartan_data::num_records");
    DEBUG_RETURN(number_del_records);
}

/* read header from file */
int Spartan_data::read_header()
{
    int i;
    int len;

    DEBUG_ENTER("Spartan_data::read_header");
    if (number_records == -1)
    {
        my_seek(data_file, 0L, MY_SEEK_SET, MYF(0));
        i = my_read(data_file, (byte *)&crashed, sizeof(bool), MYF(0));
        i = my_read(data_file, (byte *)&len, sizeof(int), MYF(0));
        memcpy(&number_records, &len, sizeof(int));
        i = my_read(data_file, (byte *)&len, sizeof(int), MYF(0));
        memcpy(&number_del_records, &len, sizeof(int));
    }
    else
        my_seek(data_file, header_size, MY_SEEK_SET, MYF(0));
    DEBUG_RETURN(0);
}

/* write header to file */
int Spartan_data::write_header()
{
    int i;

    DEBUG_ENTER("Spartan_data::write_header");
    if (number_records != -1)
    {
        my_seek(data_file, 0L, MY_SEEK_SET, MYF(0));
        i = my_write(data_file, (byte *)&crashed, sizeof(bool), MYF(0));
        i = my_write(data_file, (byte *)&number_records, sizeof(int), MYF(0));
        i = my_write(data_file, (byte *)&number_del_records, sizeof(int), MYF(0));
    }
    DEBUG_RETURN(0);
}

/* get position of the data file */
long long Spartan_data::cur_position()
{
    long long pos;

    DEBUG_ENTER("Spartan_data::cur_position");
    pos = my_seek(data_file, 0L, MY_SEEK_CUR, MYF(0));
    if (pos == 0)
        DEBUG_RETURN(header_size);
}
```



```

    DEBUG_RETURN(pos);
}

/* truncate the data file */
int Spartan_data::trunc_table()
{
    DEBUG_ENTER("Spartan_data::trunc_table");
    if (data_file != -1 )
    {
        my_chsize(data_file, 0, 0, MYF(MY_WME));
        write_header();
    }
    DEBUG_RETURN(0);
}

/* determine the row size of the data file */
int Spartan_data::row_size(int length)
{
    DEBUG_ENTER("Spartan_data::row_size");
    DEBUG_RETURN(length + record_header_size);
}

```

请注意我用来存储数据的格式。这个类是为了支持从磁盘读出数据并把内存里的数据写入磁盘而设计的。我使用了一个字节指针来分配用于存储行的内存块。这样就可以把表里的行以MySQL的内部行格式写入磁盘了。同样，只要让handler类指向将从优化器返回的内存块，就可按照MySQL的内部行格式从磁盘读出数据并把它写入一个内存缓冲区。

不过，使用字节指针来分配用于存储行的内存块也带来了一个问题：我无法准确地预知存储一个行需要多少内存，而存储引擎遇到的表有时会包含长度可变的字段，甚至是BLOB字段。为了解决这个问题，我决定在每个行的开头加上一个整数类型的长度字段来记录这个行的长度。这样一来，当扫描一个文件并读取长度可变的行时，可以先读出长度字段，再按照它的值把字节读到内存缓冲区里。

提示 在为MySQL服务器编写一个扩展模块的时候，你应该坚持使用MySQL AB公司提供的my_XXX()形式的工具方法。my_XXX()形式的工具方法分别封装着许多基本的操作系统函数，并提供了更好的跨平台支持。

虽然源代码比较长，但这个数据类相当简明，它可以用来实现一个存储引擎所需要的基本的读、写操作。不过，我还想让Spartan存储引擎更高效。为了让数据文件获得更好的性能，需要添加一种索引机制。事情从这里开始变得复杂起来了。

2. Spartan_index类

为了解决给数据文件建立索引的问题，我另外实现了一个名为Spartan_index的索引类。这个索引类的基本功能包括：加快“点查询”（point query，利用索引直接检索某个特定的记录）和“区间查询”（range query，按递增或递减顺序读取一组记录）的速度，把索引数据缓存在内存里以加快索引搜索速度等。代码清单7-5完整地给出了Spartan_index类的头文件的源代码。

代码清单7-5 Spartan_index类的头文件 (Spartan_index.h)

```

/*
    spartan_index.h

    This header file defines a simple index class that can
    be used to store file pointer indexes (long long). The
    class keeps the entire index in memory for fast access.
    The internal memory structure is a linked list. While
    not as efficient as a B-tree, it should be usable for
    most testing environments. The constructor accepts the
    max key length. This is used for all nodes in the index.

    File Layout:
        SOF                                max_key_len (int)
        SOF + sizeof(int)                  crashed (bool)
        SOF + sizeof(int) + sizeof(bool) DATA BEGINS HERE
*/
#include "my_global.h"
#include "my_sys.h"

const long METADATA_SIZE = sizeof(int) + sizeof(bool);
/*
    This is the node that stores the key and the file
    position for the data row.
*/
struct SDE_INDEX
{
    byte *key;
    long long pos;
    int length;
};
/* defines (doubly) linked list for internal list */
struct SDE_NDX_NODE
{
    SDE_INDEX key_ndx;
    SDE_NDX_NODE *next;
    SDE_NDX_NODE *prev;
};

class Spartan_index
{
public:
    Spartan_index(int keylen);
    Spartan_index();
    ~Spartan_index(void);
    int open_index(char *path);
    int create_index(char *path, int keylen);
    int insert_key(SDE_INDEX *ndx, bool allow_dupes);
    int delete_key(byte *buf, long long pos, int key_len);

```

```

long long get_index_pos(byte *buf, int key_len);
long long get_first_pos();
byte *get_first_key();
byte *get_last_key();
byte *get_next_key();
byte *get_prev_key();
int close_index();
int load_index();
int destroy_index();
SDE_INDEX *seek_index(byte *key, int key_len);
SDE_NDX_NODE *seek_index_pos(byte *key, int key_len);
int save_index();
int trunc_index();
private:
    File index_file;
    int max_key_len;
    SDE_NDX_NODE *root;
    SDE_NDX_NODE *range_ptr;
    int block_size;
    bool crashed;
    int read_header();
    int write_header();
    long long write_row(SDE_INDEX *ndx);
    SDE_INDEX *read_row(long long Position);
    long long curfpos();
};

```

这个类给出了创建、打开、关闭、读和写方法的预期形式。load_index()方法将把整个索引文件读入内存，并把索引保存为一个双向链表。所有的索引扫描和引用方法将访问内存中的那个链表而不是访问磁盘。这可以节省大量的时间，且提供了一种把整个索引保留在内存里以加快插入和删除操作的办法。相应的，save_index()方法将把索引从内存写回磁盘。这两个方法应该这样来使用：在打开表的时候调用load_index()方法，在关闭表的时候调用save_index()方法。

你也许想知道，这个办法有没有长度方面的限制。是这样的，根据每个索引项的长度、索引项的个数和有多少个索引，这种实现可能会有一些限制。但就这个教程和Spartan存储引擎可以预见的用途而言，这不会成为一个问题。

另一个值得探讨的问题是应不应该使用双向链表。如果你需要的是一种高速的索引存储机制，这种数据结构应该不是你的首选。你可能会选择使用一个B树或它的某种变体来创建一个更高效的索引访问方法。我之所以会选用双向链表，是因为它不仅容易使用，还可以让代码比较容易维护。毕竟这个例子是为了演示如何为存储引擎添加索引功能——不是为了演示如何通过编程实现一个B树结构。正是因为链表容易通过编程实现，Spartan_index类的代码才那么简明易懂。就这个教程的目的而言，链表结构的性能已经足够好了。在创建你自己的存储引擎时，可以这样来做：先用Spartan_index类来搭建存储引擎，等存储引擎的其他部分全部弄好之后，再把注意力转移到为它提供一个更好的索引类上来。

代码清单7-6给出了Spartan_index类完整的源代码。这份清单相当长，如果你现在没有时间把它通读一遍，以后再找个时间研究那些方法也不迟。现在介绍如何构建Spartan存储引擎。

代码清单7-6 Spartan_index类的源代码 (Spartan_index.cpp)

```
/*
Spartan_index.cpp

This class reads and writes an index file for use with the Spartan data
class. The file format is a simple binary storage of the
Spartan_index::SDE_INDEX structure. The size of the key can be set via
the constructor.
*/
#include "spartan_index.h"
#include <my_dir.h>

/* constuctor takes the maximum key length for the keys */
Spartan_index::Spartan_index(int keylen)
{
    root = NULL;
    crashed = false;
    max_key_len = keylen;
    index_file = -1;
    block_size = max_key_len + sizeof(long long) + sizeof(int);
}
/* constuctor (overloaded) assumes existing file */
Spartan_index::Spartan_index()
{
    root = NULL;
    crashed = false;
    max_key_len = -1;
    index_file = -1;
    block_size = -1;
}

/* destructor */
Spartan_index::~Spartan_index(void)
{
}

/* create the index file */
int Spartan_index::create_index(char *path, int keylen)
{
    DEBUG_ENTER("Spartan_index::create_index");
    open_index(path);
    max_key_len = keylen;
    /*
    Block size is the key length plus the size of the index
    length variable.
    */
    block_size = max_key_len + sizeof(long long);
    write_header();
    DEBUG_RETURN(0);
}
```



```

}

/* open index specified as path (pat+filename) */
int Spartan_index::open_index(char *path)
{
    DEBUG_ENTER("Spartan_index::open_index");
    /*
     * Open the file with read/write mode,
     * create the file if not found,
     * treat file as binary, and use default flags.
     */
    index_file = my_open(path, O_RDWR | O_CREAT | O_BINARY | O_SHARE, MYF(0));
    if(index_file == -1)
        DEBUG_RETURN(errno);
    read_header();
    DEBUG_RETURN(0);
}

/* read header from file */
int Spartan_index::read_header()
{
    int i;
    byte len;

    DEBUG_ENTER("Spartan_index::read_header");
    if (block_size == -1)
    {
        /*
         * Seek the start of the file.
         * Read the maximum key length value.
         */
        my_seek(index_file, 0l, MY_SEEK_SET, MYF(0));
        i = my_read(index_file, &len, sizeof(int), MYF(0));
        memcpy(&max_key_len, &len, sizeof(int));
        /*
         * Calculate block size as maximum key length plus
         * the size of the key plus the crashed status byte.
         */
        block_size = max_key_len + sizeof(long long) + sizeof(int);
        i = my_read(index_file, &len, sizeof(bool), MYF(0));
        memcpy(&crashed, &len, sizeof(bool));
    }
    else
    {
        i = (int)my_seek(index_file, sizeof(int) + sizeof(bool), MY_SEEK_SET, MYF(0));
    }
    DEBUG_RETURN(0);
}

/* write header to file */
int Spartan_index::write_header()

```

```
{
    int i;
    byte len;

    DEBUG_ENTER("Spartan_index::write_header");
    if (block_size != -1)
    {
        /*
         * Seek the start of the file and write the maximum key length
         * then write the crashed status byte.
         */
        my_seek(index_file, 0l, MY_SEEK_SET, MYF(0));
        memcpy(&len, &max_key_len, sizeof(int));
        i = my_write(index_file, &len, sizeof(int), MYF(0));
        memcpy(&len, &crashed, sizeof(bool));
        i = my_write(index_file, &len, sizeof(bool), MYF(0));
    }
    DEBUG_RETURN(0);
}

/* write a row (SDE_INDEX struct) to the index file */
long long Spartan_index::write_row(SDE_INDEX *ndx)
{
    long long pos;
    int i;
    int len;

    DEBUG_ENTER("Spartan_index::write_row");
    /*
     * Seek the end of the file (always append)
     */
    pos = my_seek(index_file, 0l, MY_SEEK_END, MYF(0));
    /*
     * Write the key value.
     */
    i = my_write(index_file, ndx->key, max_key_len, MYF(0));
    memcpy(&pos, &ndx->pos, sizeof(long long));
    /*
     * Write the file position for the key value.
     */
    i = i + my_write(index_file, (byte *)&pos, sizeof(long long), MYF(0));
    memcpy(&len, &ndx->length, sizeof(int));
    /*
     * Write the length of the key.
     */
    i = i + my_write(index_file, (byte *)&len, sizeof(int), MYF(0));
    if (i == -1)
        pos = i;
    DEBUG_RETURN(pos);
}
```

```

}

/* read a row (SDE_INDEX struct) from the index file */
SDE_INDEX *Spartan_index::read_row(long long Position)
{
    int i;
    long long pos;
    SDE_INDEX *ndx = NULL;

    DEBUG_ENTER("Spartan_index::read_row");
    /*
     * Seek the position in the file (Position).
     */
    pos = my_seek(index_file, (ulong) Position, MY_SEEK_SET, MYF(0));
    if (pos != -1L)
    {
        ndx = new SDE_INDEX();
        /*
         * Read the key value.
         */
        i = my_read(index_file, ndx->key, max_key_len, MYF(0));
        /*
         * Read the key value. If error, return NULL.
         */
        i = my_read(index_file, (byte *)&ndx->pos, sizeof(long long), MYF(0));
        if (i == -1)
        {
            delete ndx;
            ndx = NULL;
        }
    }
    DEBUG_RETURN(ndx);
}

/* insert a key into the index in memory */
int Spartan_index::insert_key(SDE_INDEX *ndx, bool allow_dupes)
{
    SDE_NDX_NODE *p = NULL;
    SDE_NDX_NODE *n = NULL;
    SDE_NDX_NODE *o = NULL;
    int i = -1;
    int icmp;
    bool dupe = false;
    bool done = false;

    DEBUG_ENTER("Spartan_index::insert_key");
    /*
     * If this is a new index, insert first key as the root node.
     */
    if (root == NULL)

```

```

{
    root = new SDE_NDX_NODE();
    root->next = NULL;
    root->prev = NULL;
    memcpy(root->key_ndx.key, ndx->key, max_key_len);
    root->key_ndx.pos = ndx->pos;
    root->key_ndx.length = ndx->length;
}
else //set pointer to root
    p = root;
/*
    Loop through the linked list until a value greater than the
    key to be inserted, then insert new key before that one.
*/
while ((p != NULL) && !done)
{
    icmp = memcmp(ndx->key, p->key_ndx.key,
                  (ndx->length > p->key_ndx.length) ?
                  ndx->length : p->key_ndx.length);
    if (icmp > 0) // key is greater than current key in list
    {
        n = p;
        p = p->next;
    }
    /*
        If dupes not allowed, stop and return NULL
    */
    else if (!allow_dupes && (icmp == 0))
    {
        p = NULL;
        dupe = true;
    }
    else
    {
        n = p->prev; //stop, insert at n->prev
        done = true;
    }
}
/*
    If position found (n != NULL) and dupes permitted,
    insert key. If p is NULL insert at end else insert in middle
    of list.
*/
if ((n != NULL) && !dupe)
{
    if (p == NULL) //insert at end
    {
        p = new SDE_NDX_NODE();
        n->next = p;
        p->prev = n;
    }

```



```

    memcpy(p->key_ndx.key, ndx->key, max_key_len);
    p->key_ndx.pos = ndx->pos;
    p->key_ndx.length = ndx->length;
}
else
{
    o = new SDE_NDX_NODE();
    memcpy(o->key_ndx.key, ndx->key, max_key_len);
    o->key_ndx.pos = ndx->pos;
    o->key_ndx.length = ndx->length;
    o->next = p;
    o->prev = n;
    n->next = o;
    p->prev = o;
}
i = 1;
}
DEBUG_RETURN(i);
}

/* delete a key from the index in memory. Note:
   position is included for indexes that allow dupes */
int Spartan_index::delete_key(byte *buf, long long pos, int key_len)
{
    SDE_NDX_NODE *p;
    int icmp;
    int buf_len;
    bool done = false;

    DEBUG_ENTER("Spartan_index::delete_key");
    p = root;
    /*
       Search for the key in the list. If found, delete it!
    */
    while ((p != NULL) && !done)
    {
        buf_len = p->key_ndx.length;
        icmp = memcmp(buf, p->key_ndx.key,
                      (buf_len > key_len) ? buf_len : key_len);
        if (icmp == 0)
        {
            if (pos != -1)
            {
                if (pos == p->key_ndx.pos)
                {
                    done = true;
                }
                else
                {
                    done = true;
                }
            }
            else
            {
                p = p->next;
            }
        }
    }
}

```

```

if (p != NULL)
{
    /*
     * Reset pointers for deleted node in list.
     */
    if (p->next != NULL)
        p->next->prev = p->prev;
    if (p->prev != NULL)
        p->prev->next = p->next;
    else
        root = p->next;
    delete p;
}
DEBUG_RETURN(0);
}

/* update key in place (so if key changes!) */
int Spartan_index::update_key(byte *buf, long long pos, int key_len)
{
    SDE_NDX_NODE *p;
    bool done = false;

    DEBUG_ENTER("Spartan_index::update_key");
    p = root;
    /*
     * Search for the key.
     */
    while ((p != NULL) && !done)
    {
        if (p->key_ndx.pos == pos)
            done = true;
        else
            p = p->next;
    }
    /*
     * If key found, overwrite key value in node.
     */
    if (p != NULL)
    {
        memcpy(p->key_ndx.key, buf, key_len);
    }
    DEBUG_RETURN(0);
}

/* get the current position of the key in the index file */
long long Spartan_index::get_index_pos(byte *buf, int key_len)
{
    long long pos = -1;

    DEBUG_ENTER("Spartan_index::get_index_pos");
    SDE_INDEX *ndx;

```

```

    ndx = seek_index(buf, key_len);
    if (ndx != NULL)
        pos = ndx->pos;
    DEBUG_RETURN(pos);
}

/* get next key in list */
byte *Spartan_index::get_next_key()
{
    byte *key = 0;

    DEBUG_ENTER("Spartan_index::get_next_key");
    if (range_ptr != NULL)
    {
        key = (byte *)my_malloc(max_key_len, MYF(MY_ZEROFILL | MY_WME));
        memcpy(key, range_ptr->key_ndx.key, range_ptr->key_ndx.length);
        range_ptr = range_ptr->next;
    }
    DEBUG_RETURN(key);
}

/* get prev key in list */
byte *Spartan_index::get_prev_key()
{
    byte *key = 0;

    DEBUG_ENTER("Spartan_index::get_prev_key");
    if (range_ptr != NULL)
    {
        key = (byte *)my_malloc(max_key_len, MYF(MY_ZEROFILL | MY_WME));
        memcpy(key, range_ptr->key_ndx.key, range_ptr->key_ndx.length);
        range_ptr = range_ptr->prev;
    }
    DEBUG_RETURN(key);
}

/* get first key in list */
byte *Spartan_index::get_first_key()
{
    SDE_NDX_NODE *n = root;
    byte *key = 0;

    DEBUG_ENTER("Spartan_index::get_first_key");
    if (root != NULL)
    {
        key = (byte *)my_malloc(max_key_len, MYF(MY_ZEROFILL | MY_WME));
        memcpy(key, n->key_ndx.key, n->key_ndx.length);
    }
    DEBUG_RETURN(key);
}

```

```
/* get last key in list */
byte *Spartan_index::get_last_key()
{
    SDE_NDX_NODE *n = root;
    byte *key = 0;

    DEBUG_ENTER("Spartan_index::get_last_key");
    while (n->next != NULL)
        n = n->next;
    if (n != NULL)
    {
        key = (byte *)my_malloc(max_key_len, MYF(MY_ZEROFILL | MY_WME));
        memcpy(key, n->key_ndx.key, n->key_ndx.length);
    }
    DEBUG_RETURN(key);
}

/* just close the index */
int Spartan_index::close_index()
{
    SDE_NDX_NODE *p;

    DEBUG_ENTER("Spartan_index::close_index");
    if (index_file != -1)
    {
        my_close(index_file, MYF(0));
        index_file = -1;
    }
    while (root != NULL)
    {
        p = root;
        root = root->next;
        delete p;
    }
    DEBUG_RETURN(0);
}

/* find a key in the index */
SDE_INDEX *Spartan_index::seek_index(byte *key, int key_len)
{
    SDE_INDEX *ndx = NULL;
    SDE_NDX_NODE *n = root;
    int buf_len;
    bool done = false;

    DEBUG_ENTER("Spartan_index::seek_index");
    if (n != NULL)
    {
        while((n != NULL) && !done)
        {

```



```

        buf_len = n->key_ndx.length;
        if (memcmp(n->key_ndx.key, key,
            (buf_len > key_len) ? buf_len : key_len) == 0)
            done = true;
        else
            n = n->next;
    }
}
if (n != NULL)
{
    ndx = &n->key_ndx;
    range_ptr = n;
}
DEBUG_RETURN(ndx);
}

/* find a key in the index and return position too */
SDE_NDX_NODE *Spartan_index::seek_index_pos(byte *key, int key_len)
{
    SDE_NDX_NODE *n = root;
    int buf_len;
    bool done = false;
    DEBUG_ENTER("Spartan_index::seek_index_pos");
    if (n != NULL)
    {
        while((n->next != NULL) && !done)
        {
            buf_len = n->key_ndx.length;
            if (memcmp(n->key_ndx.key, key,
                (buf_len > key_len) ? buf_len : key_len) == 0)
                done = true;
            else if (n->next != NULL)
                n = n->next;
        }
    }
    DEBUG_RETURN(n);
}

/* read the index file from disk and store in memory */
int Spartan_index::load_index()
{
    SDE_INDEX *ndx;
    int i = 0;

    DEBUG_ENTER("Spartan_index::load_index");
    if (root != NULL)
        destroy_index();
    /*
    First, read the metadata at the front of the index.
    */

```

```
read_header();
while(!eof(index_file))
{
    ndx = new SDE_INDEX();
    i = my_read(index_file, (byte *)&ndx->key, max_key_len, MYF(0));
    i = my_read(index_file, (byte *)&ndx->pos, sizeof(long long), MYF(0));
    i = my_read(index_file, (byte *)&ndx->length, sizeof(int), MYF(0));
    insert_key(ndx, false);
}
DEBUG_RETURN(0);
}

/* get current position of index file */
long long Spartan_index::curfpos()
{
    long long pos = 0;
    DEBUG_ENTER("Spartan_index::curfpos");
    pos = my_seek(index_file, 0l, MY_SEEK_CUR, MYF(0));
    DEBUG_RETURN(pos);
}

/* write the index back to disk */
int Spartan_index::save_index()
{
    SDE_NDX_NODE *n = root;
    int i;

    DEBUG_ENTER("Spartan_index::save_index");
    i = chsize(index_file, 0l);
    write_header();
    while (n != NULL)
    {
        write_row(&n->key_ndx);
        n = n->next;
    }
    DEBUG_RETURN(0);
}

int Spartan_index::destroy_index()
{
    SDE_NDX_NODE *n = root;

    DEBUG_ENTER("Spartan_index::destroy_index");
    while (root != NULL)
    {
        n = root;
        root = n->next;
        delete n;
    }
    root = NULL;
}
```

```

    DEBUG_RETURN(0);
}

/* ket the file position of the first key in index */
long long Spartan_index::get_first_pos()
{
    long long pos = -1;

    DEBUG_ENTER("Spartan_index::get_first_pos");
    if (root != NULL)
        pos = root->key_ndx.pos;
    DEBUG_RETURN(pos);
}

/* truncate the index file */
int Spartan_index::trunc_index()
{
    DEBUG_ENTER("Spartan_index::trunc_table");
    if (index_file != -1)
    {
        my_chsize(index_file, 0, 0, MYF(MY_WME));
        write_header();
    }
    DEBUG_RETURN(0);
}

```

请注意，类似于Spartan_data类的情况，我使用DEBUG例程来用于调试的跟踪元素。还用到了my_xxx()平台安全工具方法。

提示 my_xxx()形式的工具方法可以在MySQL源代码树跟目录的mysys目录里找到。它们一般是一些存储在同名文件里的C函数，比如说，my_write()方法位于my_write.c文件里。

每个索引项由三个部分构成：一个指向一个内存块的字节指针，用来保存键；一个位置值（long long），用来保存相关记录在磁盘上的偏移地址，Spartan_data类将利用这个偏移地址来设置它的文件指针；一个长度字段，用来存放键本身的长度。length变量将在内存比较方法里用来设置比较长度。这些数据项保存在名为SDE_INDEX的结构里，双向链表的结点是另一个包含着一个SDE_INDEX结构的结构。链表结点结构的名称是SDE_NDX_NODE，也为链表提供了next和prev指针。

在使用索引来保存Spartan_data类文件里的数据存放位置时，你可以调用insert_index()方法，需要把键和数据项在文件里的偏移地址值传递给这个方法。这个偏移值是由my_write()方法返回的。这个技巧可以让你直接把指向数据的索引指针的值保存到磁盘并重新使用该信息，而无需把它转换为一个供磁盘文件读写指针使用的值。

Spartan_index类生成的索引在磁盘上将被保存为一个连续的数据块，这个数据块里的每个数据项长度固定为SDE_INDEX结构的长度。这个文件有一个首部，它用来保存一个崩溃状态变量和一个键最大长度变量。崩溃状态变量可以让你对某个文件是否受到了破坏或是否有错误发生迅速做出判断。这类情况非常罕见，但我们在编程时不能不把它考虑进去。至于为什么要把Spartan_index类生成的索引项设置为固定长度，而不是像Spartan_data类那样使用一个可变长度的字段，是因为这可以简化磁盘读、

写方法的编程工作。这是一种以空间换效率的做法，从实际效果看，我的决定还算是明智的。

到了这一步，创建一个存储引擎时最苦最累的工作（编写底层I/O函数）就告一段落了。接下来，请随我一起去看看如何通过编程来构建一个基本的存储引擎。在把Spartan存储引擎从阶段1完善到阶段5的过程中，我们还会反复多次地回到Spartan_data和Spartan_index类这里来。

7.2.2 预备知识

以下内容假设已经配置好了开发环境，并在调试模式下编译了MySQL服务器（参见第5章）。你将经历创建Spartan存储引擎的每一个阶段。在出发之前，还需要做一项非常重要的准备工作：创建一个用来测试Spartan存储引擎的测试文件，从而让开发工作的每个阶段都有一个明确的目标。我已经在本书的第4章对MySQL测试工具包，以及如何创建和运行一个测试的步骤进行过专题介绍。如果你们有什么不明白的地方，请参考那一章的相关内容。

提示 MySQL测试工具包目前还无法在Windows平台上使用。不过，Windows用户可以通过剪贴操作把有关语句复制到MySQL客户程序里来创建必要的测试文件。有了测试文件，运行它们就很简单了。

你们应该做的第一件事情是为Spartan存储引擎编写一个测试文件。遵循测试驱动开发技术的理念，虽然这个存储引擎还不存在，你仍应该在开始编写代码前为它创建一个测试。现在就来做这件事。

这个测试文件应该从一个简单的测试开始：Spartan存储引擎能不能创建表并从中检索出行。当然可以一口气把本章后面内容里的测试全都编写到测试文件里，但我认为还是从一个简单的测试开始并随着Spartan存储引擎的逐步完善而分阶段地进行扩展更好。这么做的好处是：你的测试将只对当前阶段进行，不会因为那些尚未实现的功能而报告出错。代码清单7-7给出了一个可以用来对阶段1里的Spartan存储引擎进行基本测试的样板测试文件。

随着这个教程的展开，我们将把新的语句分阶段地添加到这个文件里；等Spartan存储引擎完成的时候，将得到一个完整的测试集。

代码清单7-7 Spartan存储引擎的测试文件（spartandb.test）

```
#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings

CREATE TABLE t1 (
  col_a int,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;

DROP TABLE t1;
```

可以把这个文件创建在MySQL源代码树根目录下的/mysql-test/t子目录里。当第一次执行的时候，报告出错是很正常的。事实上，也应该在开始阶段1之前执行一次这个测试，要不怎么知道这个

测试能不能工作呢？如果还记得我在第4章里是怎么做的，就应该知道从/mysql-test子目录发出下面这些命令就可以运行这个测试：

```
%> touch r/spartandb.result
%> ./mysql-test-run.pl spartanddb
%> cp r/cab.reject r/spartandb.result
%> ./mysql-test-run.pl spartandb
```

试试看，报告出错了吗？这次的测试结论应该是[failed]，但如果去查看它生成的日志文件，却不会看到任何错误。这是怎么回事？是这样的，如果你在CREATE TABLE语句里指定的存储引擎不存在，MySQL就将使用默认的存储引擎。具体到这个例子，MySQL服务器会发出一条警告消息，说它将使用默认的MyISAM存储引擎；这很正常，因为我们还没把Spartan存储引擎创建出来呢。代码清单7-8给出了一个/mysql-test/r/spartandb.log日志文件的例子。

代码清单7-8 测试日志文件示例

```
drop table if exists t1;
CREATE TABLE t1 (
  col_a int,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;
Warnings:
Warning 1266 Using storage engine MyISAM for table 't1'
DROP TABLE t1;
```

7.2.3 阶段 1：生成引擎存根

这个阶段的目标是产生一个生成方法存根的存储引擎。该生成方法存根的引擎将支持最基本的操作：允许我们在CREATE语句里选择存储引擎，并且能够为基表创建元文件(*.frm)。这听起来很简单，做起来也不太难。但我想提醒大家的是：虽然阶段1里的存储引擎还不能存储任何数据^①，但一个成功的阶段1存储引擎能够把它自己注册到MySQL服务器里，这要求你必须把所有需要修改的MySQL源代码都修改好。我在前面曾经说过，有些改动在未来的MySQL版本里有可能不再需要；但就目前而言，需要我们修改的地方还真不少。作为一条经验，在开始修改MySQL源代码之前，你最好先到MySQL AB公司的网站上查一下你那个版本的《MySQL参考手册》有没有什么新内容。

1. 创建Spartan源文件

你需要做的第一件事是在MySQL源代码树根目录下的/storage子目录里创建一个名为spartan的子目录。我将以MySQL自带的示例存储引擎为基础来创建Spartan存储引擎，《MySQL参考手册》也建议人们使用示例存储引擎的源代码作为基础。示例存储引擎包含着各种必要的方法，更重要的是MySQL AB公司保证它们都是用正确的代码语句实现出来的。这就使得我们为Spartan存储引擎创建基本源文件的工作变得容易。

提示 绝大多数源文件在Linux平台上被命名为*.cc，在Windows平台上被命名为*.cpp。

^① 可以帮助人们开发存储引擎的资料非常稀少，就算有几本这方面的书籍，它们给出的例子也往往仅限于阶段1引擎。我就是因为对此深有体会才编写了本章的内容。

从/storage/example子目录把*.cc和*.h文件复制到/storge/spartan子目录。你的spartan子目录现在应该有两个文件：ha_example.cc（在Windows平台上是ha_example.cpp）和ha_example.h。ha_前缀的含义是这些文件是从handler类派生出来的，它代表着一个表处理机（table handler）。请把ha_example.cc（在Windows平台上是ha_example.cpp）、ha_example.h文件重新命名为ha_spartan.cc（在Windows平台上是ha_spartan.cpp）和ha_spartan.h。

注解 表处理机（table handler）这个名词现在已被替换为存储引擎（storage engine），但你们仍会在MySQL的文档和许多书刊里看到表处理机这个词，它们是同义词。

下一步是把那两个源文件里的example和EXAMPLE分别替换为spartan和SPARTAN。你可以用最喜欢的代码编辑器或文本处理器来快速完成这些替换。这将把所有标识符里的example字样替换为spartan（比如说，st_example_share将变成st_spartan_share）。千万注意，这些替换操作需要区分字母的大小写；如果你弄错了，存储引擎将无法工作。

接下来，编辑ha_spartan.cc文件，把handlerton声明里的注释改过来。具体修改位置如下所示。

```
handlerton spartan_hton= {
    MYSQL_HANDLERTON_INTERFACE_VERSION,
    "SPARTAN",
    SHOW_OPTION_YES,
    "Spartan storage engine",
    DB_TYPE_SPARTAN_DB,
    ...
}
```

最后，编辑ha_spartan.h文件，添加一条include语句把spartan_data.h文件包括进来，如下所示：

```
#include "spartan_data.h"
```

(1) 在Linux平台中把源文件添加到项目文件里

如果你使用的是Linux，将需要创建一个制作文件并修改MySQL源代码树根目录里的configure脚本。从/storage/example子目录把Makefile.am文件复制到/storge/spartan子目录。打开Makefile.am文件，先把所有的example替换为spartan，再把spartan_data.h和spartan_data.cc文件分别添加到noinst_HEADERS和libspartan_a_SOURCES语句里，如下所示。

```
noinst_HEADERS = ha_spartan.h spartan_data.h
libspartan_a_SOURCES = ha_spartan.cc spartan_data.cc
```

接下来，从/storage/example子目录把Makefile.in文件复制到/storge/spartan子目录。打开Makefile.in文件，先把所有的example替换为spartan，再把spartan_data.h和spartan_data.cc文件分别添加到noinst_HEADERS和libspartan_a_SOURCES语句里，如下所示。

```
noinst_HEADERS = ha_spartan.h spartan_data.h
libspartan_a_SOURCES = ha_spartan.cc spartan_data.cc
am_libspartan_a_OBJECTS = ha_spartan.$(OBJEXT) spartan_data.$(OBJEXT)
```

下一步是修改MySQL源代码树根目录里的configure脚本。打开这个文件并搜索单词csv。第一个匹配位置应该在Optional Packages:节里。请像下面这样把选项语句--with-spartan-storage-engine添加进去。

```
--with-csv-storage-engine
    enable csv storage engine (default is "yes")
--with-spartan-storage-engine
    enable spartan storage engine (default is "yes")
--with-blackhole-storage-engine
    enable blackhole storage engine (default is no)
```

添加这些语句的最佳办法是把CSV存储引擎的那些语句复制一份,然后把所有的csv和tina替换为spartan。接下来,将需要创建一个用来处理制作文件的节。找到CSV存储引擎的对应节,把整个节复制一份,然后进行字符串替换。替换完成后的最终的节应该是下面这样的。

```
# Check whether --with-spartan-storage-engine or
# --without-spartan-storage-engine was given.
if test "${with_spartan_storage_engine+set}" = set; then
    withval="$with_spartan_storage_engine"

else
    with_spartan_storage_engine="yes"
fi;
echo "$as_me:$LINENO: checking whether to use Spartan storage engine" >&5
echo $ECHO_N "checking whether to use Spartan storage engine... $ECHO_C" >&6
if test "${mysql_cv_use_spartan_storage_engine+set}" = set; then
    echo $ECHO_N "(cached) $ECHO_C" >&6
else
    mysql_cv_use_spartan_storage_engine=$with_spartan_storage_engine
fi
echo "$as_me:$LINENO: result: $mysql_cv_use_spartan_storage_engine" >&5
echo "${ECHO_T}$mysql_cv_use_spartan_storage_engine" >&6

if test "$mysql_cv_use_spartan_storage_engine" != no; then
if test "spartan_hton" != "no"
then
    cat >>confdefs.h <<\_ACEOF
#define WITH_spartan_storage_ENGINE 1
_ACEOF

    mysql_se_decls="${mysql_se_decls},spartan_hton"
    mysql_se_htons="${mysql_se_htons},&spartan_hton"
    if test "no" != "no"
    then
        mysql_se_objs="$mysql_se_objs no"
    fi
    mysql_se_dirs="$mysql_se_dirs storage/spartan"
    mysql_se_libs="$mysql_se_libs \$(top_builddir)/storage/spartan/libspartan.a"
else
    mysql_se_plugins="$mysql_se_plugins storage/spartan"
fi
```

```
ac_config_files="$ac_config_files storage/spartan/Makefile"
```

```
fi
```

下一个需要修改的节是`ac_config_target`。按照刚才的套路：搜索`csv`，复制一个节，进行字符串替换。替换完成后的最终的节应该是下面这样的。

```
"storage/spartan/Makefile" ) CONFIG_FILES=
"$CONFIG_FILES storage/spartan/Makefile" ;;
```

类似的，你还需要在`config.h.in`文件里添加一条`#undef`语句。打开这个文件并输入以下内容。

```
/* Build Spartan storage engine */
#undef WITH_SPARTAN_STORAGE_ENGINE
```

还需要在`configure`脚本的`configure.in`头文件里添加一条创建语句。打开这个文件并在快到这个文件末尾的地方输入以下内容（找一条现有的语句进行复制和替换）。

```
MYSQL_STORAGE_ENGINE(spartan,,,"yes",,spartan_hton,storage/spartan,no,
 \$(top_builddir)/storage/spartan/libspartan.a,[
 AC_CONFIG_FILES(storage/spartan/Makefile)
])
```

提示 上述修改在未来的MySQL版本里可能会有所变化。一定要用`./configure --help`命令去检查一下，你正在使用的源代码版本是否仍支持`--with`参数——如果有变化，请按照新的指示去修改`configure`脚本文件和`configure.in`头文件。

接下来，在`/storage/spartan`子目录里创建一个名为`.deps`的子目录，然后在`/storage/spartan/.deps`子目录里创建一个名为`ha_example.Po`的文件。把这个文件的内容设置为`#dummy`。等你编译这些代码的时候，编译工具会用Spartan存储引擎的正确参数覆盖这个文件。

完成上述修改之后，还需要再修改几个服务器文件，才能运行`configure`脚本。至于如何编译这些代码的问题，应该等到把Spartan存储引擎添加到MySQL服务器里以后再做考虑。

(2) 在Windows平台中把源文件添加到项目文件里

如果你使用的是Windows，将需要创建一个新项目，把源文件添加到新项目里，对新项目进行必要的配置。打开MySQL源代码树根目录里的主解决方案文件。为这个解决方案添加一个名为`spartan`的新项目。把项目文件保存到`/storage/spartan`子目录。你应该创建一个C++ Win32 ► Win32 Project项目，千万不要错选成`console`（控制台）或`.NET`项目。在Win32 Application Wizard（Win32应用程序向导）里，单击Next按钮，然后把应用程序类型设置为静态库并关闭（弃选）各种预编译的首部。在Visual Studio把新项目创建出来之后，把`Spartan_data`和`Spartan_index`类的源文件添加到新项目里。

为了让`spartan`项目找到它所需要的包含文件，需要打开Spartan Property Pages对话框，在左窗口里依次展开C/C++和General分支，然后在右窗口里选中Additional Include Directories选项。这里有个偷懒的办法：打开Example Property Pages对话框，从那里把相应的配置字符串复制到`spartan`项目里来。图7-2是完成上述修改后的Spartan Property Pages对话框的样子。

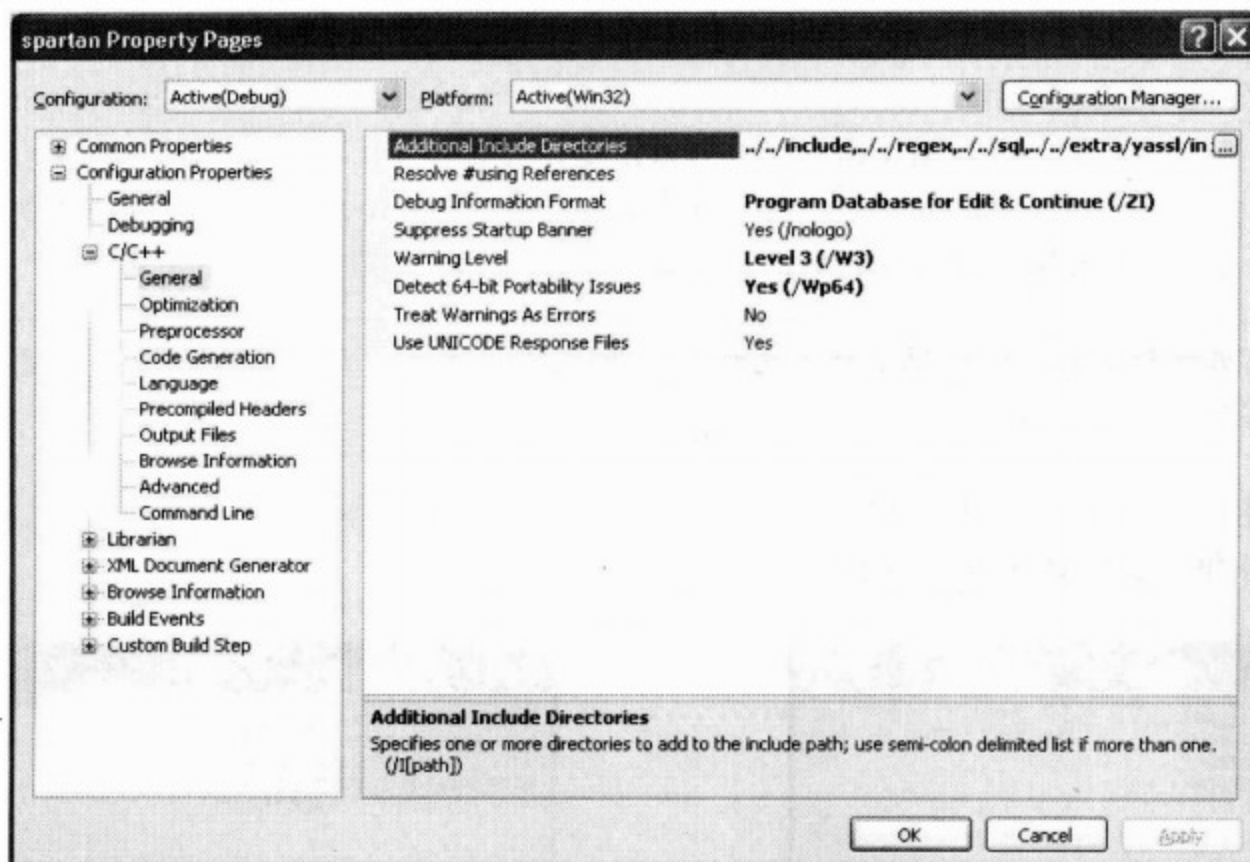


图7-2 选择Additional Include Directories选项

接下来，还是在这个对话框里，在左窗口里依次展开C/C++和General分支，然后在右窗口里把Runtime Library选项设置为Multi-threaded debug (/MTd)。图7-3是完成上述修改后的Spartan Property Pages对话框的样子。在确认没有错误之后，请关闭这个对话框。

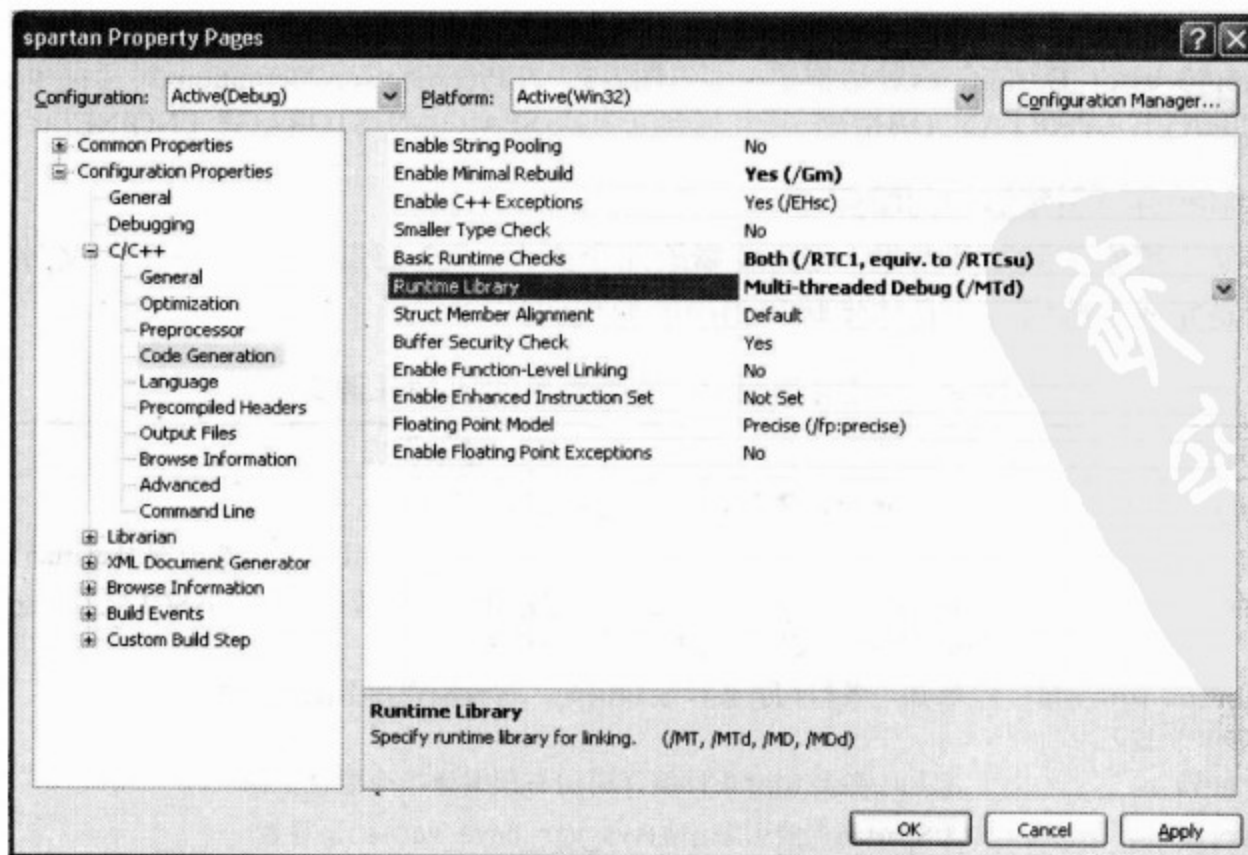


图7-3 把Runtime Library选项修改为Multi-threaded debug (/MTd)

还需要通过Project ► Project Dependencies菜单把spartan项目添加到mysqld项目里去。一定要把编译模式设置为“调试”。最后一步是修改mysqld项目的项目设置。你需要添加HAVE_SPARTAN_DB和WITH_SPARTAN_STORAGE_ENGINE预编译指令，具体做法是：打开mysqld项目的项目属性对话框，在左窗口里依次展开C/C++和Preprocessor分支，然后在右窗口里单击Preprocessor Definitions选项旁边的省略号按钮（...）。图7-4和图7-5是完成上述修改后的窗口画面（那些参数的顺序无关紧要）。

提示 如果打算编辑的某个源代码呈暗灰色的不可选状态，其原因可能是相应的预编译指令没有给出或拼写错误。

完成上述修改之后，还需要再修改几个服务器文件才能编译你的MySQL服务器。应该等到把Spartan存储引擎添加到MySQL服务器里以后再进行编译。

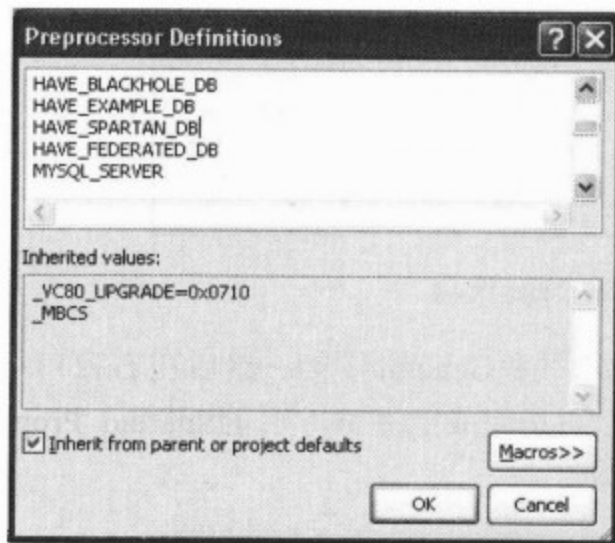


图7-4 在Preprocessor Definitions对话框里添加HAVE_SPARTAN_DB指令

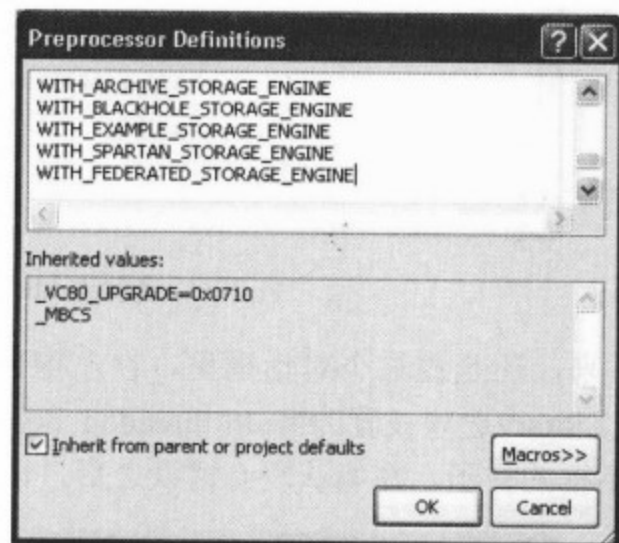


图7-5 在Preprocessor Definitions对话框里添加WITH_SPARTAN_STORAGE_ENGINE指令

2. 把Spartan存储引擎添加到服务器

为了把存储引擎添加到服务器里，还需要修改几个文件。表7-2列出了需要修改的文件和需要修改的内容，表后是完成这些修改的具体步骤和相应的源代码片段。

表7-2 创建存储引擎：需要修改的MySQL源文件

源文件	修改情况
/include/my_config.h	为Spartan存储引擎添加一条#define语句
/sql/handler.h	为legacy_db_type枚举集合添加一个元素，让系统能够识别出spartan表类型
/sql/handler.cc	在show_table_alias_st结构里添加一条新数据项，让MySQL能够把spartan字符串与handler.h文件里的表类型标识符关联起来
/sql/handler-nt-win.cpp (仅适用于Windows平台)	添加一条#ifdef条件编译语句，与预编译指令配合工作
/sql/mysql_priv.h	添加一条#ifdef条件编译语句与预编译指令配合工作
/sql/set_var.cc	为Spartan存储引擎添加sys_var_have_variable设置
/sql/mysqld.cc	为HAVE_SPARTAN_DB定义添加#undef语句和SHOW_COMP_OPTION选项

打开my_config.h文件并搜索example。应该在接近这个文件末尾的地方找到示例存储引擎的#define语句。复制并粘贴那两条语句，然后为Spartan存储引擎进行字符串替换。下面是完成上述修改后的代码片段。

```
/* Build Spartan storage engine */
#define WITH_SPARTAN_STORAGE_ENGINE 1
```

打开handler.h文件并修改legacy_db_type枚举集合的定义。在列表的末尾，在DEFAULT数据库类型元素的上面，加上DB_TYPE_SPARTAN_DB元素。这是为了保证分配给DB_TYPE_SPARTAN_DB元素的值不会与默认的存储引擎元素（DB_TYPE_DEFAULT=127）发生冲突。下面是完成上述修改后的代码片段。

```
enum legacy_db_type
{
    DB_TYPE_UNKNOWN=0, DB_TYPE_DIAB_ISAM=1,
    DB_TYPE_HASH, DB_TYPE_MISAM, DB_TYPE_PISAM,
    DB_TYPE_RMS_ISAM, DB_TYPE_HEAP, DB_TYPE_ISAM,
    DB_TYPE_MRG_ISAM, DB_TYPE_MYISAM, DB_TYPE_MRG_MYISAM,
    DB_TYPE_BERKELEY_DB, DB_TYPE_INNODB,
    DB_TYPE_GEMINI, DB_TYPE_NDBCLUSTER,
    DB_TYPE_EXAMPLE_DB, DB_TYPE_ARCHIVE_DB, DB_TYPE_CSV_DB,
    DB_TYPE_FEDERATED_DB,
    DB_TYPE_BLACKHOLE_DB,
    DB_TYPE_PARTITION_DB,
    DB_TYPE_BINLOG,
    DB_TYPE_SPARTAN_DB,
    DB_TYPE_DEFAULT=127 // Must be last
};
```

打开handler.cc文件并修改show_table_alias_st数组。在列表的末尾，在UNKNOWN数据库类型元素的上面，加上DB_TYPE_SPARTAN_DB元素和字符串。这些数据项的顺序并不重要，但MySQL AB公司惯于使用最后一个元素作为“岗哨”，所以应该把它留在那里别动。下面是完成上述修改后的代码片段。

```
struct show_table_alias_st sys_table_aliases[]=
{
    {"INNOBASE", DB_TYPE_INNODB},
    {"NDB", DB_TYPE_NDBCLUSTER},
    {"BDB", DB_TYPE_BERKELEY_DB},
    {"HEAP", DB_TYPE_HEAP},
    {"MERGE", DB_TYPE_MRG_MYISAM},
    {"SPARTAN", DB_TYPE_SPARTAN_DB},
    {Nulls, DB_TYPE_UNKNOWN}
};
```

打开handler-win.cpp文件并把#ifdef语句和extern语句添加到这个文件里（仅适用于Windows平台）。从example存储引擎那里把相应的#ifdef语句复制并粘贴过来，然后进行字符串替换。下面是完成上述修改后的代码片段。

```
#ifdef WITH_SPARTAN_STORAGE_ENGINE
extern handlerton spartan_hnton;
#endif
```

你还需要修改sys_table_types结构并为Spartan存储引擎添加一条#ifdef语句（仅适用于Windows

平台)。从示例存储引擎那里把相应的`#ifdef`语句复制并粘贴过来, 然后进行字符串替换。下面是完成上述修改后的代码片段。

```
handler_ton *sys_table_types[]=
{
    &heap_pton,
    &myisam_pton,
    ...
#ifdef WITH_SPARTAN_STORAGE_ENGINE
    &spartan_pton,
#endif
```

打开`mysql_priv.h`文件并为Spartan存储引擎添加一条`#ifdef`语句。从示例存储引擎那里把相应的`#ifdef`语句复制并粘贴过来, 然后进行字符串替换。下面是完成上述修改后的代码片段。

```
#ifdef WITH_SPARTAN_STORAGE_ENGINE
extern handler_ton spartan_pton;
#define have_spartan_db spartan_pton.state
#else
extern SHOW_COMP_OPTION have_spartan_db;
#endif
```

打开`set_var.cc`文件并为Spartan存储引擎添加一个`sys_var_have_variable`数组。从示例存储引擎那里把相应的语句复制并粘贴过来, 然后进行字符串替换。下面是完成上述修改后的代码片段。

```
sys_var_have_variable sys_have_spartan_db("have_spartan_engine",
                                           &have_spartan_db);
```

你还需要在`init_vars`数组里添加一条`sys_have_spartan_db.name`定义。从示例存储引擎那里把相应的语句复制并粘贴过来, 然后进行字符串替换。下面是完成上述修改后的代码片段。

```
SHOW_VAR init_vars[] = {
    {"auto_increment_increment", (char*) &sys_auto_increment_increment, SHOW_SYS},
    {"auto_increment_offset", (char*) &sys_auto_increment_offset, SHOW_SYS},
    ...
    {sys_have_example_db.name, (char*) &have_example_db, SHOW_HAVE},
    {sys_have_spartan_db.name, (char*) &have_spartan_db, SHOW_HAVE},
    {sys_have_federated_db.name, (char*) &have_federated_db, SHOW_HAVE},
    ...
}
```

最后一个需要修改的文件是`mysqld.cc`。打开这个文件并为Spartan存储引擎添加一条`#undef`语句。你还需要在那些`#undef`语句后面添加一个`SHOW_COMP_OPTION`选项。从示例存储引擎那里把相应的语句复制并粘贴过来, 然后进行字符串替换。下面是完成上述修改后的代码片段。

```
/*
Instantiate have_xxx for missing storage engines
*/
#ifdef WITH_SPARTAN_STORAGE_ENGINE
#define HAVE_SPARTAN_STORAGE_ENGINE
#endif
#ifdef HAVE_BERKELEY_DB
#define HAVE_BERKELEY_DB
#endif
#ifdef HAVE_INNO_DB
#define HAVE_INNO_DB
#endif
#ifdef HAVE_NDB_CLUSTER
#define HAVE_NDB_CLUSTER
#endif
#ifdef HAVE_EXAMPLE_DB
#define HAVE_EXAMPLE_DB
#endif
#ifdef HAVE_SPARTAN_DB
#define HAVE_SPARTAN_DB
#endif
```



```
#undef have_archive_db
#undef have_csv_db
#undef have_federated_db
#undef have_partition_db
#undef have_blackhole_db

SHOW_COMP_OPTION have_berkeley_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_innodb= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_ndbcluster= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_example_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_spartan_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_archive_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_csv_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_federated_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_partition_db= SHOW_OPTION_NO;
SHOW_COMP_OPTION have_blackhole_db= SHOW_OPTION_NO;
```

还有一个地方需要修改。在ha_spartan.cc文件的末尾，应该看到一个mysql_declare_plugin节。这部分代码负责实现插入接口的“热”插入（即插即用）功能。你可以在这个节里输入一些自己认为与Spartan存储引擎有关的信息，例如自己的名字和编程体会等。这个节目前还派不上用场，等MySQL AB公司把插件式存储引擎的体系结构完全建立好以后，你将需要通过这个节启用插入接口。

注解 这个结构最有可能发生变化。请随时注意MySQL网站上的在线《MySQL参考手册》里的最新修改。

```
#ifdef MYSQL_PLUGIN
mysql_declare_plugin
{
    MYSQL_STORAGE_ENGINE_PLUGIN,
    &spartan_hton,
    spartan_hton.name,
    "Dr. Bell",
    "Spartan Storage Engine -- Expert MySQL",
    spartan_init_func, /* Plugin Init */
    spartan_done_func, /* Plugin Deinit */
    0x0001 /* 0.1 */,
}
mysql_declare_plugin_end;
#endif
```

你肯定会觉得需要修改的地方实在太多了——对，是这样的。幸好，MySQL AB公司已经承诺在未来的MySQL版本里改善这种状况。

3. 编译Spartan存储引擎

把上面提到的那些修改工作全部完成之后，就可以编译MySQL服务器，和测试Spartan存储引擎了。这个过程与编译其他程序没什么两样。为了生成踪迹文件和使用一个交互式调试器在服务器运行时调试有关的源代码，你应该在调试模式下编译MySQL服务器。

(1) 在Linux平台上进行编译

在Linux平台上编译MySQL服务器需要使用configure、make和make install命令来建立这个项目。要想让MySQL服务器能够识别Spartan存储引擎并在调试模式下进行编译，将需要执行如下所示的命令：

```
../configure --with-spartan-storage-engine --with-debug
make
make install
```

所有的相关项目会根据需要被自动编译进来。因为我们修改了几个关键的头文件并添加了几条新的预编译指令，这次编译可能需要多花点儿时间。

(2) 在Windows平台上进行编译

在Windows平台上编译MySQL服务器，需要在Visual Studio里建立mysqld项目，所有的相关项目会根据需要被自动编译进来。因为我们修改了几个关键的头文件并添加了几条新的预编译指令，这次编译可能需要多花点儿时间。

4. 测试Spartan存储引擎（阶段1）

编译好MySQL服务器之后，就可以启动它运行了。你肯定很想用交互式MySQL客户端程序去试试刚编译出来的服务器，这没什么问题，因为我当初就是这么想和这么做的。代码清单7-9是通过MySQL客户端程序执行了一些SQL命令后得到的结果。这个例子执行了SHOW STORAGE ENGINES、CREATE TABLE、SHOW CREATE TABLE和DROP TABLE命令。从这些命令的返回结果看，它们都执行成功了，这意味着我将要进行的spartan测试应该顺利通过。

代码清单7-9 以手动方式测试Spartan存储引擎（阶段1）示例

```
mysql> SHOW STORAGE ENGINES;
```

Engine	Support	Comment	Transactions	XA	Savepoints
EXAMPLE	YES	Example storage engine	NO	NO	NO
MEMORY	YES	Hash based, stored in me	NO	NO	NO
MRG_MYISAM	YES	Collection of identical	NO	NO	NO
MyISAM	DEFAULT	Default engine as of MyS	NO	NO	NO
BLACKHOLE	YES	/dev/null storage engine	NO	NO	NO
SPARTAN	YES	Spartan storage engine	NO	NO	NO
InnoDB	YES	Supports transactions, r	YES	YES	YES
ARCHIVE	YES	Archive storage engine	NO	NO	NO
FEDERATED	YES	Federated MySQL storage	YES	NO	NO

```
9 rows in set (0.02 sec)
```

```
mysql> USE test;
```

```
Database changed
```

```
mysql> CREATE TABLE t1 (col_a int, col_b varchar(20), col_c int) ENGINE=SPARTAN;
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> SHOW CREATE TABLE t1 \G
```

Current database: test

***** 1. row *****

```
Table: t1
Create Table: CREATE TABLE `t1` (
  `col_a` int(11) DEFAULT NULL,
  `col_b` varchar(20) DEFAULT NULL,
  `col_c` int(11) DEFAULT NULL
) ENGINE=SPARTAN DEFAULT CHARSET=latin1
1 row in set (0.20 sec)
```

```
mysql> DROP TABLE t1;
```

```
Query OK, 0 rows affected (1 min 19.14 sec)
```

```
mysql>
```

因为Spartan存储引擎出现在了SHOW STORAGE ENGINES和SHOW CREATE TABLE命令的执行结果里，我知道它已经开始工作了。这个引擎真的能连接上服务器吗？这一点从SHOW命令的执行结果里是看不出来的。还好，一旦出现这种情况，CREATE TABLE命令就会告诉我们它使用的是MyISAM存储引擎而不是Spartan存储引擎。

如果你使用的是Linux平台，还应该运行一下之前创建的spartandb测试。这次运行这个测试的时候，测试能通过。这是因为Spartan存储引擎现在已经成为MySQL服务器的一部分并能够被识别出来。接下来，请给这个测试添加一条SELECT命令，然后再运行它一次。这次的测试也应该能通过。既然如此，你就可以把最新的测试结果添加到/r子目录里并开始一次自动测试了。代码清单7-10是改进后的测试。

代码清单7-10 已更新的Spartan存储引擎的测试文件（spartandb.test）

```
#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings

CREATE TABLE t1 (
  col_a int,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;

SELECT * FROM t1;

DROP TABLE t1;
```

Spartan存储引擎的第1阶段开发工作就此圆满结束。它已经能成功地插入MySQL服务器并等我们去为它添加Spartan_data和Spartan_index类。在下一阶段，将为它添加创建、打开、关闭和删除文件

的能力。这听起来好像没有多大进展，但事情总要一步一步地做，每添加一项功能，都应该进行测试和调试，直到没有任何问题时，才可以开始实现更复杂的操作。

7.2.4 阶段 2：处理表

这个阶段的目标是产生一个能够创建、打开、关闭和删除数据文件的生成方法存根的存储引擎。在这个阶段，你需要创建一些基本的文件处理例程并确保存储引擎能够正确地完成各种必要的文件操作。MySQL已经为我们准备了一系列文件I/O例程，它们对底层函数进行了封装，在跨平台使用时不容易引起兼容问题。下面是这些函数的一部分，更多细节可以在/mysys子目录中的文件里查到。

- ❑ my_create (...): 创建文件。
- ❑ my_open (...): 打开文件。
- ❑ my_read (...): 从文件读出数据。
- ❑ my_write (...): 把数据写入文件。
- ❑ my_delete (...): 删除文件。
- ❑ fn_format (...): 创建一条可以跨平台的路径语句。

在这个阶段，我将演示如何使用Spartan_data类来处理底层I/O。详细介绍每一处需要修改的地方并在完成每处修改后给出有关函数的完整源代码。

1. 更新Spartan存储引擎的源文件

需要做的第一件事情是从Apress出版公司的网站下载Spartan类的源代码压缩文件并把它们复制到/storage/spartan子目录，或者使用本章前面你自行创建的spartan_data.cc和spartan_data.h文件。

因为我将使用Spartan_data类来处理底层I/O，需要创建一个对象指针来容纳这个类的一个实例。我需要把它放到一个能被共享的地方，因为不想允许两个或更多个Spartan_data类的实例同时访问同一个文件。（这是可以做到的，但会使这个例子变得非常复杂，不仅需要额外做许多事情，也偏离了本章的讨论重点。）具体地说，我将把一个对象引用指针放到Spartan类的st_spartan_share结构里去。

提示 你应该每完成一处修改就编译一次spartan项目，这可以让你及时发现自己做的修改有没有错误。在修改下一个地方之前，一定要保证所有的错误都改正过来了。

(1) 更新头文件

打开ha_spartan.h文件并添加一条#include语句来引入spartan_data.h头文件，然后在st_spartan_share结构里添加一个对象引用指针。代码清单7-11是完成这个修改后的代码片段（为简明起见，我删节了代码中的注释）。完成这个修改后，要重新编译spartan项目的源文件以确保没有任何错误。

代码清单7-11 修改ha_spartan.h文件里的st_spartan_share结构

```
#include "spartan_data.h"

#ifdef USE_PRAGMA_INTERFACE
#pragma interface      /* gcc class implementation */
#endif
```



```
...
typedef struct st_spartan_share {
    char *table_name;
    uint table_name_length, use_count;
    pthread_mutex_t mutex;
    THR_LOCK lock;
    Spartan_data *data_class;
} SPARTAN_SHARE;
```

(2) 更新类文件

接下来的修改都发生在ha_spartan.cc文件里。打开这个文件并找到get_share()方法。因为我们在ha_spartan.h文件里给st_spartan_share结构添加了一个对象引用指针，所以get_share()方法在初始化share变量时，将需要为该指针提供一个对象实例，这就需把Spartan_data类的实例化代码添加到get_share()方法里去。我给那个对象引用指针起的名字是data_class。代码清单7-12是完成这个修改后的get_share()方法的代码片段。请注意，share结构使用的计数器（share->use_count）的初始值是0，每添加一个引用，这个计数器将加上一个1；每减少一个引用，这个计数器将减去一个1；当这个计数器的值重新变成0时，分配给share结构的内存就将被释放。这将确保share结构在还有代码在使用它时不会被释放。这很有必要，因为包含在share结构里的数据和索引是全局共享的。

提示 如果你使用的是Windows平台，可能会遇到Visual Studio里的IntelliSense无法识别Spartan_data类的问题。此时，你将需要修复.ncb文件。具体做法是：退出Visual Studio，删除MySQL源代码根目录里的.ncb文件，然后重新编译mysqld项目。这可能需要花费一些时间，但在完成编译后，IntelliSense就应该能用了。

代码清单7-12 修改ha_spartan.cc文件里的get_share()方法

```

        &tmp_name, length+1,
        NULLS))
    {
        pthread_mutex_unlock(&spartan_mutex);
        return NULL;
    }
    /*
     * Set the initial variables to defaults.
     */
    share->use_count=0;
    share->table_name_length=length;
    share->table_name = (char *)my_malloc(length + 1, MYF(0));
    strcpy(share->table_name, table_name);
    /*
     * Insert table name into hash for future reference.
     */
    if (my_hash_insert(&spartan_open_tables, (byte*) share))
        goto error;
    thr_lock_init(&share->lock);
    /*
     * Create an instance of data class
     */
    share->data_class = new Spartan_data();
    pthread_mutex_init(&share->mutex, MY_MUTEX_INIT_FAST);
}
share->use_count++; // increment use count on reference
pthread_mutex_unlock(&spartan_mutex); //release mutex lock
return share;

error:
    pthread_mutex_destroy(&share->mutex);
    my_free((gptr) share, MYF(0));

    return NULL;
}

```

相应的，我们还需要在释放share结构的时候把那个对象引用指针也释放掉。请找到free_share()方法并把释放data_class对象引用指针的代码添加进去。代码清单7-13是完成这个修改后的free_share()方法的代码片段。

代码清单7-13 修改ha_spartan.cc文件里的free_share()方法

```

static int free_share(SPARTAN_SHARE *share)
{
    DBUG_ENTER("ha_spartan::free_share");
    pthread_mutex_lock(&spartan_mutex);
    if (--share->use_count)
    {
        if (share->data_class != NULL)
            delete share->data_class;
    }
}

```

```

share->data_class = NULL;
/*
  Remove the share from the hash.
*/
hash_delete(&spartan_open_tables, (byte*) share);
thr_lock_delete(&share->lock);
pthread_mutex_destroy(&share->mutex);
my_free((gptr)share->table_name, MYF(0));
}
pthread_mutex_unlock(&spartan_mutex);

DEBUG_RETURN(0);
}

```

Spartan存储引擎的handler实例还需要为数据文件提供文件扩展名。因为有一个数据文件和一个索引文件，所以你需要创建两个文件扩展名。请定义两个文件扩展名并把它们添加到ha_spartan_exts数组里。我为数据文件定义的扩展名是.sde，为索引文件定义的扩展名是.sdi。MySQL需要使用这些扩展名来完成文件删除等文件管理操作。找到ha_spartan_exts数组，在它前面添加一条#define语句，再把你定义的文件扩展名添加到这个数组里去。代码清单7-14是完成这个修改后的代码片段。

代码清单7-14 修改ha_spartan.cc文件里的ha_spartan_exts数组

```

#define SDE_EXT ".sde"
#define SDI_EXT ".sdi"
...
static const char *ha_spartan_exts[] = {
    SDE_EXT,
    SDI_EXT,
    NullS
};

```

需要添加的第一个操作是“创建文件”操作。这个操作将创建一个空白文件来容纳表里的数据。找到create()方法，把用来获得share结构一份副本的代码添加到其中，调用data_class对象的create_table()方法，最后关闭表。代码清单7-15给出了完成这个修改后的create()方法。我将在阶段5演示如何添加一个索引类。

代码清单7-15 修改ha_spartan.cc文件里的create()方法

```

int ha_spartan::create(const char *name, TABLE *table_arg,
                      HA_CREATE_INFO *create_info)
{
    DEBUG_ENTER("ha_spartan::create");
    char name_buff[FN_REFLLEN];

    if (!(share = get_share(name, table)))
        DEBUG_RETURN(1);
    /*
     Call the data class create table method.
     Note: the fn_format() method correctly creates a file name from the
     name passed into the method.

```

```

*/
if (share->data_class->create_table(fn_format(name_buff, name, "", SDE_EXT,
                                             MY_REPLACE_EXT|MY_UNPACK_FILENAME)))
    DEBUG_RETURN(-1);
share->data_class->close_table();
}

```

你需要添加的第二个操作是“打开文件”操作。这个操作将打开一个包含着表数据的文件。找到 `open()` 方法，把用来获得 `share` 结构一份副本的代码添加到其中，然后打开表。代码清单7-16给出了完成这个修改后的 `open()` 方法。我将在阶段5演示如何添加一个索引类。

代码清单7-16 修改 `ha_spartan.cc` 文件里 `open()` 方法

```

int ha_spartan::open(const char *name, int mode, uint test_if_locked)
{
    DEBUG_ENTER("ha_spartan::open");
    char name_buff[FN_REFLen];
    if (!(share = get_share(name, table)))
        DEBUG_RETURN(1);
    /*
     * Call the data class open table method.
     * Note: the fn_format() method correctly creates a file name from the
     * name passed into the method.
     */
    share->data_class->open_table(fn_format(name_buff, name, "", SDE_EXT,
                                             MY_REPLACE_EXT|MY_UNPACK_FILENAME));
    thr_lock_data_init(&share->lock, &lock, NULL);
    DEBUG_RETURN(0);
}

```

请注意，我在这段代码里用 `pthread_mutex_lock(&spartan_mutex)` 和 `pthread_mutex_unlock(&spartan_mutex)` 方法调用划出了一个关键节（critical section）。我这么做是因为 `Spartan_data` 类的对象只有一个实例，而我想限制每次只能有一个线程进入这个关键节去访问这个对象。这一限制对所有操作（例如读取数据）来说并无必要，但却是一个好的做法。

需要添加的第三个操作是“关闭文件”。不过，因为 `free_share()` 方法会在释放内存后自动完成这个操作，你已经用不着再添加什么东西了。

你需要添加的下一个操作是“删除文件”。这个操作将删除一个包含着表数据的文件。找到 `delete_table()` 方法，把用来获得 `share` 结构一份副本的代码添加到其中，关闭表，最后调用 `my_delete()` 函数删除表。代码清单7-17给出了完成这个修改后的 `delete_table()` 方法。我将在阶段5演示如何添加一个索引类。

代码清单7-17 修改 `ha_spartan.cc` 文件里的 `delete_table()` 方法

```

int ha_spartan::delete_table(const char *name)
{
    DEBUG_ENTER("ha_spartan::delete_table");
    char name_buff[FN_REFLen];

```



```

/*
    Begin critical section by locking the spartan mutex variable.
*/
pthread_mutex_lock(&spartan_mutex);
if (!(share = get_share(name, table)))
    DEBUG_RETURN(1);
share->data_class->close_table();
/*
    Call the mysql delete file method.
    Note: the fn_format() method correctly creates a file name from the
    name passed into the method.
*/
my_delete(fn_format(name_buff, name, "", SDE_EXT,
                    MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
/*
    End section by unlocking the spartan mutex variable.
*/
pthread_mutex_unlock(&spartan_mutex);
DEBUG_RETURN(0);
}

```

事情还没有全部结束，还有一个操作需要添加，粗心大意的程序员往往会忘记它。“RENAME TABLE”命令允许用户重命名表，存储引擎也必须能够把文件复制到一个新名字并删除旧文件。重新命名*.frm文件的操作MySQL服务器会替我们完成，但数据文件的复制工作还要由自己来安排。找到rename_table()方法，把用来获得share结构的一份副本的代码添加到其中，关闭表，最后调用my_copy()函数复制表。代码清单7-18给出了完成这个修改后的rename_table()方法。我将在阶段5演示如何添加一个索引类。

代码清单7-18 修改ha_spartan.cc文件里rename_table()方法

```

int ha_spartan::rename_table(const char * from, const char * to)
{
    DEBUG_ENTER("ha_spartan::rename_table ");
    char data_from[FN_REFLen];
    char data_to[FN_REFLen];

    if (!(share = get_share(from, table)))
        DEBUG_RETURN(1);
    /*
        Begin critical section by locking the spartan mutex variable.
    */
    pthread_mutex_lock(&spartan_mutex);
    /*
        Close the table then copy it then reopen new file.
    */
    share->data_class->close_table();
    my_copy(fn_format(data_from, from, "", SDE_EXT,
                    MY_REPLACE_EXT|MY_UNPACK_FILENAME),
            fn_format(data_to, to, "", SDE_EXT,

```

```

        MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
share->data_class->open_table(data_to);
/*
    End section by unlocking the spartan mutex variable.
*/
pthread_mutex_unlock(&spartan_mutex);
/*
    Delete the file using MySQL's delete file method.
*/
my_delete(data_from, MYF(0));
DEBUG_RETURN(0);
}

```

好了，现在就实现了一个完整的阶段2存储引擎。可以编译MySQL服务器并运行有关的测试了。

2. 测试Spartan存储引擎（阶段2）

当再次运行spartandb测试的时候，应该看到所有的语句都执行成功了。不过，有两件事是这个测试无法替你验证的。第一，需要确认*.sde文件真的被创建和删除过。第二，需要确认rename命令真的能用。

用来创建和删除一个表的命令很容易测试。启动你的MySQL服务器和一个MySQL客户端程序，在客户端程序里执行spartandb测试文件里的CREATE语句，然后用你的文件管理器进入/data/test文件夹。应该在那里看到两个文件：t1.frm和t1.sde。接下来，回到MySQL客户端程序，发出一条DROP命令，然后回到/data/test文件夹，看那两个文件是不是真的被删除了。

用来重新命名一个表的命令也很容易测试。先重复一次刚才的CREATE语句，然后输入下面这条命令：

```
RENAME TABLE t1 TO t2;
```

用文件管理器进入/data/test文件夹。应该在那里看到两个文件：t2.frm和t2.sde。接下来，回到MySQL客户端程序，发出一条DROP命令，然后回到/data/test文件夹看那两个文件是不是真的被删除了。

在确认RENAME语句执行无误之后，把它添加到spartandb测试文件里再进行一次全面测试。这次测试应该不会报告任何错误。代码清单7-19给出了修改后的spartandb.test文件。

代码清单7-19 更新Spartan存储引擎的测试文件（spartandb.test）

```

#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings

CREATE TABLE t1 (
    col_a int,
    col_b varchar(20),
    col_c int
) ENGINE=SPARTAN;

SELECT * FROM t1;

```

```
RENAME TABLE t1 TO t2;
```

```
DROP TABLE t2;
```

Spartan存储引擎的第2阶段开发工作就此圆满结束。在第一阶段的基础上，它现在还可以完成文件的创建、打开、关闭、删除和重命名操作。下一阶段将为它添加读、写数据的能力。

7.2.5 阶段 3：数据的读/写

这个阶段的目标是在前两个阶段的基础上为存储引擎添加读、写数据的能力。在这个阶段，我将向读者演示如何使用Spartan_data类来读、写数据。我将详细介绍每一处需要修改的地方并在完成每处修改后给出有关函数的完整源代码。

1. 更新Spartan的源文件

把存储引擎从阶段2提升到阶段3需要对基本的数据读、写例程进行改进。具体到这个例子，为了实现读操作，需要修改ha_spartan.oc文件里的rnd_init()、rnd_next()、position()和rnd_pos()方法。position()和rnd_pos()方法用来对大量数据进行排序，它们都使用了一个内部缓冲区来存储行。为了实现写操作，你只需要修改write_row()方法。

(1) 更新头文件

position()方法需要你保存一个指针——它或者是一个记录偏移位置，或者是一个将被用在排序操作里的键值。MySQL AB公司提供了一个很巧妙的办法来解决这个问题，你过一会儿就会在position()方法里见到它。首先，打开ha_spartan.h文件，在ha_spartan类里添加一个current_position变量。代码清单7-20是完成这个修改后的代码片段。

代码清单7-20 修改ha_spartan.h文件里的ha_spartan类

```
class ha_spartan: public handler
{
    THR_LOCK_DATA lock;      /* MySQL lock */
    SPARTAN_SHARE *share;    /* Shared lock info */
    off_t current_position; /* Current position in the file during a file scan */
    ...
}
```

(2) 更新源文件

接下来的修改都发生在ha_spartan.cc文件里。需要修改的第一个方法是rnd_init()，你需要在这里为一次全表扫描操作设置初始条件。具体到这个例子，需要把当前位置设置为0（文件的开头），把记录个数(records变量)设置为0，再给出你想为排序操作使用的数据项的长度。具体到这个例子，应该使用一个long long，因为那是文件里的当前读写位置的数据类型。代码清单7-21是完成这个修改后的rnd_init()方法的代码片段。

代码清单7-21 修改ha_spartan.cc文件里的rnd_init()方法

```
int ha_spartan::rnd_init(bool scan)
{
    DEBUG_ENTER("ha_spartan::rnd_init");
    current_position = 0;
    records = 0;
}
```

```

    ref_length = sizeof(long long);
    DEBUG_RETURN(0);
}

```

注意 从这一小节开始，我们将给Spartan存储引擎添加的功能都是示例存储引擎不具备的（示例存储引擎只实现到了阶段2），所以千万不要把返回代码写错了。示例引擎在遇到它不支持的函数时会执行一条DEBUG_RETURN(HA_ERR_WRONG_COMMAND);语句向优化器返回一个“错误命令”代码。千万记得把这些返回代码改成其他的东西（比如0）。

接下来需要修改的方法是rnd_next()，它负责从文件里读出下一条记录并检查文件结束（end-of-file）标志。在这个方法里，可以调用数据类（本例中的Spartan_data类）的read_row()方法来完成这些工作，将需要把记录缓冲区的名字、这个缓冲区的长度和文件里的当前读写位置传递给read_row()方法。请注意这段代码是如何检查eof标记和统计记录个数的。这个方法还把当前读写位置保存到了current_position变量里，这样一来，这个方法的下一次调用就会去读取文件里的下一条记录了。代码清单7-22是完成这个修改后rnd_init()文件的代码片段。

代码清单7-22 修改ha_spartan.cc文件里的rnd_next()方法

```

int ha_spartan::rnd_next(byte *buf)
{
    int rc;

    DEBUG_ENTER("ha_spartan::rnd_next");
    ha_statistic_increment(&SSV::ha_read_rnd_next_count);
    /*
     * Read the row from the data file.
     */
    rc = share->data_class->read_row(buf, table->s->rec_buff_length,
                                     current_position);
    if (rc != -1)
        current_position = (off_t)share->data_class->cur_position();
    else
        DEBUG_RETURN(HA_ERR_END_OF_FILE);
    records++;
    DEBUG_RETURN(0);
}

```

我刚才讲过，Spartan_data类将按照MySQL内部缓冲区里的格式把记录存入文件，但它会在每条记录的前面加上几个字节作为标头，这些字节包括一个删除标志和该记录的长度（用于扫描和修复）。如果你想让存储引擎以另外一种格式来存储数据，将需要在这里对数据进行必要的格式转换。你们可以在ha_tina.cc文件里找到一个进行这种转换的例子，它的核心部分如下所示。

```

for (Field **field=table->field ; *field ; field++)
{
    /* copy field data to your own storage type */
    my_value = (*field)->val_str();
    my_store_field(my_value);
}

```


这条for语句将遍历field数组并把数据转换为另一种格式后写入文件。ha_tina::find_current_row()方法里也有一个类似的例子。

下一个需要修改的方法是position(),它负责把文件的当前读写位置记录到MySQL的指针存储机制里。每个rnd_next()调用的后面都应该有一个position()调用。用来保存和检索这些指针的方法是my_store_ptr()和my_get_ptr()。my_store_ptr()方法需要三个输入参数:一个引用变量(你想把东西保存到此处)、你想保存的数据长度、你想保存的东西。my_get_ptr()会接受两个输入参数(一个引用变量和你想检索的数据的长度),并返回你想检索的数据。如果你想按照某种顺序对行进行排序,就肯定会用到这两个函数。请好好看看代码清单7-23给出的position()方法,尤其要注意它是如何调用my_store_ptr()方法的。

代码清单7-23 修改ha_spartan.cc文件里的position()方法

```
void ha_spartan::position(const byte *record)
{
    DEBUG_ENTER("ha_spartan::position");
    my_store_ptr(ref, ref_length, current_position);
    DEBUG_VOID_RETURN;
}
```

还需要修改rnd_pos()方法,需要用它来检索被保存起来的当前位置并从那个位置读入一个行。请注意,这个方法还对读操作计数器ha_read_rnd_next_count进行了递增。这个信息可以让优化器对表里有多少个行大致有个数并据此对以后的查询命令进行优化。代码清单7-24是完成这个修改后的rnd_pos()方法的代码片段。

代码清单7-24 修改ha_spartan.cc文件里的rnd_pos()方法

```
int ha_spartan::rnd_pos(byte * buf, byte *pos)
{
    DEBUG_ENTER("ha_spartan::rnd_pos");
    ha_statistic_increment(&SSV::ha_read_rnd_next_count);
    current_position = (off_t)my_get_ptr(pos, ref_length);
    share->data_class->read_row(buf, current_position, -1);
    DEBUG_RETURN(0);
}
```

接着需要修改info()方法,它向优化器返回的信息可以帮助优化器选择一个最优的执行路径。这是一个很有意思的方法,它的源代码里的注释很幽默。需要在这个方法里做的事情是对记录个数进行统计,并返回一个总数。不过,你没有必要真的进行统计,因为MySQL AB公司建议程序员应该让这个方法总是返回一个不小于2的数字——如果你告诉优化器说你的表只包含一个行,优化器将不会对它进行某些优化。代码清单7-25是完成修改后的info()方法。

代码清单7-25 修改ha_spartan.cc文件里的info()方法

```
void ha_spartan::info(uint flag)
{
    DEBUG_ENTER("ha_spartan::info");
    /* This is a lie, but you don't want the optimizer to see zero or 1 */
```

```

    if (records < 2)
        records= 2;
    DEBUG_VOID_RETURN;
}

```

最后一个需要修改的方法是write_row(), 你需要在这个方法里使用Spartan_data类把数据写入数据文件。我刚才讲过, Spartan_data类只需把记录缓冲区里的内容(加上一个删除标志和该记录的长度)直接写到磁盘上就完成了写操作。代码清单7-26是完成修改后的write_row()方法。

代码清单7-26 修改ha_spartan.cc文件里的write_row()方法

```

int ha_spartan::write_row(byte * buf)
{
    DEBUG_ENTER("ha_spartan::write_row");
    ha_statistic_increment(&SSV::ha_write_count);
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->write_row(buf, table->s->rec_buff_length);
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}

```

请注意, 我又一次为写方法设置了mutex(比如关键节), 这是为了确保不会有两个线程同时向同一个文件写数据。完成上述这些修改后, 编译MySQL服务器并调试之。调试工作完成后, 你就有了一个阶段3存储引擎。剩下的事情就是编译MySQL服务器并对它进行测试了。

2. 测试Spartan存储引擎(阶段3)

首先, 再次运行spartandb测试, 你应该看到所有的语句都执行成功了。我为什么要让你在每次测试开始的时候先把上次的测试再运行一遍呢? 因为这可以检查新添加的代码会不会对现有的代码造成不良影响。具体到这个例子, 首先应该确保自己仍能正确地创建、重新命名和删除表, 然后再对新添加的数据读、写操作进行测试。

你们新修改的这些函数很容易测试。启动刚编译好的MySQL服务器和一个MySQL客户端程序。如果你已经删除了测试表, 请先把它再次创建出来, 然后发出以下命令。

```

INSERT INTO t1 VALUES(1, "first test", 24);
INSERT INTO t1 VALUES(4, "second test", 43);
INSERT INTO t1 VALUES(3, "third test", -2);

```

这几条记录插入语句都应该没有任何错误地执行成功。万一你遇到了错误(不应该出现这种情况), 可以启动你的调试器, 在ha_spartan.cc文件里的每一个读、写函数里设置一些个断点, 然后开始调试它。用不着调试ha_spartan.cc文件以外的其他文件, 因为只有它才可能包含导致出错的代码^①。

现在, 可以发出一条SELECT语句, 并观察MySQL服务器会发回什么东西了。请输入以下命令。

```
SELECT * FROM t1;
```

你应该看到刚才插入的3条记录。代码清单7-27给出了运行这条查询命令的结果。

① 当然, 问题的根源也有可能出在底层的源代码身上——这或许是因为你刚才漏掉了什么, 也可能是因为MySQL AB公司对有关的MySQL源代码又做了修改。

代码清单7-27 运行INSERT/SELECT语句的结果

```

+-----+-----+-----+
| col_a | col_b      | col_c |
+-----+-----+-----+
| 1      | first test  | 24     |
| 4      | second test | 43     |
| 3      | third test  | -2     |
+-----+-----+-----+
3 rows in set (0.00 sec)

```

在确认数据读、写操作都能正常工作之后，把对这些操作的测试添加到spartandb测试文件里，然后再次运行全部的测试。代码清单7-28给出了修改后的spartandb.test文件。

代码清单7-28 已更新的Spartan存储引擎的测试文件（spartandb.test）

```

#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings

CREATE TABLE t1 (
  col_a int,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;

SELECT * FROM t1;
INSERT INTO t1 VALUES(1, "first test", 24);
INSERT INTO t1 VALUES(4, "second test", 43);
INSERT INTO t1 VALUES(3, "third test", -2);
SELECT * FROM t1;
RENAME TABLE t1 TO t2;
SELECT * FROM t2;
DROP TABLE t2;

```

Spartan存储引擎的第3阶段开发工作就此圆满结束。现在，在第二阶段的基础上，它还可以完成数据的读、写操作，一个基本的存储引擎已经成形了。在下一阶段，我们将为它添加更新和删除数据的能力。

7.2.6 阶段 4：数据的更新和删除

这个阶段的目标是在第3阶段的基础上为存储引擎添加更新和删除数据的功能。这个阶段演示如何使用Spartan_data类来更新和删除数据。还将详细介绍每一处需要修改的地方，并在完成每处修改后给出有关函数的完整的源代码。

在完成数据更新操作的时候，将直接用新数据覆盖掉老数据。在完成删除操作的时候，它会给你让它删除的数据加上一个“已删除”标记，并在以后的读操作里忽略那些有“已删除”标记的记录。你

需要给read_row()方法添加一些代码，好让它能跳过那些已被删除的行。这种做法似乎有浪费空间的嫌疑——如果把这个存储引擎用在有大量删除和插入操作的环境里的话，这种嫌疑还真有可能成为事实。如果你们想降低这种可能性，可以定期导出表里的数据，删除老表，然后再用导出来的数据重新创建之；这将消除那些白占用空间的记录。在设计和创建自己的存储引擎的时候，根据具体情况，你可能需要把这一点考虑进去。

1. 更新Spartan的源文件

把存储引擎从阶段3提升到阶段4需要我们对update_row()、delete_row()和delete_all_rows()方法进行改进。delete_all_rows()方法可以快速删除某个表的全部内容，而不是每次只删除一个行，这将节约不少时间。优化器可以在检测到一个用来删除大量记录的查询命令时，调用delete_all_rows()方法来完成操作。

(1) 更新头文件

把Spartan存储引擎从阶段3提升到阶段4不需要修改ha_spartan.h文件。

(2) 更新类文件

请打开ha_spartan.cc文件并找到update_row()方法。这个方法需要你把老记录和新的记录缓冲区作为参数传递给它。这对Spartan存储引擎来说是个好消息，因为我们现在还没有为它添加索引功能，它现在只能通过全表扫描操作来确定各有关记录的位置！好在Spartan_data类自带update_row()方法，它将替你完成那些工作。代码清单7-29给出了完成修改后的update_row()方法。

代码清单7-29 修改ha_spartan.cc文件里的update_row()方法

```
int ha_spartan::update_row(const byte * old_data, byte * new_data)
{
    DEBUG_ENTER("ha_spartan::update_row");
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->update_row((byte *)old_data, new_data,
                                   table->s->rec_buff_length, current_position -
                                   share->data_class->row_size(table->s->rec_buff_length));
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}
```

delete_row()方法与update_row()方法很相似。具体到这个例子，将调用Spartan_data类的delete_row()方法来完成删除操作，这个方法需要你传递以下几个参数：将被删除的行所在的缓冲区、该记录缓冲区的长度、将被删除的记录在记录缓冲区里的位置（把这个位置设置为-1将强制进行一次全表扫描）。和刚才一样，Spartan_data类里的delete_row()方法将替你完成这些工作。代码清单7-30给出了完成修改后的delete_row()方法。

代码清单7-30 修改ha_spartan.cc文件里的delete_row()方法

```
int ha_spartan::delete_row(const byte * buf)
{
    long long pos;

    DEBUG_ENTER("ha_spartan::delete_row");
    if (current_position > 0)
```



```

    pos = current_position -
        share->data_class->row_size(table->s->rec_buff_length);
else
    pos = 0;
pthread_mutex_lock(&spartan_mutex);
share->data_class->delete_row((byte *)buf,
                             table->s->rec_buff_length, pos);
pthread_mutex_unlock(&spartan_mutex);
DEBUG_RETURN(0);
}

```

需要修改的最后一个方法是delete_all_rows()。这个方法将删除给定表里的所有数据。做这件事最简单的办法是先删除相应的数据文件，再把它重新创建出来。但Spartan_data类采用的是一种稍微不同的做法：trunc_table()方法先把文件指针设置为指向数据文件的开头，然后调用my_chsize()方法截短数据文件。代码清单7-31给出了完成修改后的delete_all_row()方法。

代码清单7-31 修改ha_spartan.cc文件里的delete_all_row()方法

```

int ha_spartan::delete_all_rows()
{
    DEBUG_ENTER("ha_spartan::delete_all_rows");
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->trunc_table();
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}

```

完成上述这些修改后，编译MySQL服务器并调试之。调试工作完成后，你就有了一个阶段4存储引擎。剩下的事情就是编译MySQL服务器和对它进行测试了。

2. 测试Spartan存储引擎（阶段4）

按照惯例，在开始测试数据修改和删除操作之前，应该先确认那个阶段3引擎里的一切东西仍都工作正常。当再次运行spartan和db测试的时候，应该看到所有的语句都执行成功了。

为了测试Spartan存储引擎的数据修改和删除操作，需要提前创建一个表并在其中填充一些数据。可以像以前那样用普通的INSERT语句来填充数据，那些数据你可以随意编造，如果说有什么要求的话，那就是不要让表里的行太少。

往表里填充了一些数据之后，可随便挑选一条记录并发出如下所示的命令去修改它。

```
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 1;
```

在执行完上面这条命令之后，请用“SELECT *”命令去检查那个行是不是真的被改过来了。然后，随便挑选一条记录并发出一条如下所示的删除命令将其删除。

```
DELETE FROM t1 WHERE col_a = 3;
```

在执行完上面这条命令之后，请用SELECT *命令去检查那个行是不是真的被删除了。我们有没有漏掉什么东西？经验丰富的程序员会注意到这个测试并不完备，因为它没能覆盖Spartan_data类的所有可能性。比如说，在表的中间部分删除一个行与删除整个表的第一个和最后一个行是不一样的；数据修改操作的情况也是如此。

这没什么问题，因为你可以直接在spartan测试文件里加上这些测试项目。可以多几条INSERT语句多添加一些数据，然后对表的第一个、最后一个和中间某个行进行修改和删除。代码清单7-32给出了修改后的spartandb.test文件。

代码清单7-32 已更新的Spartan存储引擎的测试文件（spartandb.test）

```
#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings
CREATE TABLE t1 (
  col_a int,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;

SELECT * FROM t1;
INSERT INTO t1 VALUES(1, "first test", 24);
INSERT INTO t1 VALUES(4, "second test", 43);
INSERT INTO t1 VALUES(3, "fourth test", -2);
INSERT INTO t1 VALUES(4, "tenth test", 11);
INSERT INTO t1 VALUES(1, "seventh test", 20);
INSERT INTO t1 VALUES(5, "third test", 100);
SELECT * FROM t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 1;
SELECT * FROM t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 3;
SELECT * FROM t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 5;
SELECT * FROM t1;
DELETE FROM t1 WHERE col_a = 1;
SELECT * FROM t1;
DELETE FROM t1 WHERE col_a = 3;
SELECT * FROM t1;
DELETE FROM t1 WHERE col_a = 5;
SELECT * FROM t1;
RENAME TABLE t1 TO t2;
SELECT * FROM t2;
DROP TABLE t2;
```

请注意，这里插入了一些数据重复的行，因为我想看看Spartan存储引擎能不能把符合同一条件的行全部修改或删除掉。运行这个测试并查看其结果。代码清单7-33是这个测试的一个预期结果示例。当你用MySQL Test Suite工具运行这个测试的时候，它应该能圆满完成而不会出现任何错误。

代码清单7-33 Spartan存储引擎的样板测试结果（阶段4）

```
mysql> CREATE TABLE t1 (
->   col_a int,
```

```
-> col_b varchar(20),
-> col_c int
-> ) ENGINE=SPARTAN;
```

Query OK, 0 rows affected (0.22 sec)

```
mysql> SELECT * FROM t1;
```

Empty set (0.02 sec)

```
mysql> INSERT INTO t1 VALUES(1, "first test", 24);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO t1 VALUES(4, "second test", 43);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO t1 VALUES(3, "fourth test", -2);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO t1 VALUES(4, "tenth test", 11);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO t1 VALUES(1, "seventh test", 20);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> INSERT INTO t1 VALUES(5, "third test", 100);
```

Query OK, 1 row affected (0.00 sec)

```
mysql> SELECT * FROM t1;
```

col_a	col_b	col_c
1	first test	24
4	second test	43
3	fourth test	-2
4	tenth test	11
1	seventh test	20
5	third test	100

6 rows in set (0.01 sec)

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 1;
```

Query OK, 2 rows affected (0.00 sec)

Rows matched: 2 Changed: 2 Warnings: 0

```
mysql> SELECT * from t1;
```

```

+-----+-----+-----+
| col_a | col_b      | col_c |
+-----+-----+-----+
| 1      | Updated!   | 24    |
| 4      | second test| 43    |
| 3      | fourth test| -2    |
| 4      | tenth test | 11    |
| 1      | Updated!   | 20    |
| 5      | third test | 100   |
+-----+-----+-----+
6 rows in set (0.00 sec)

```

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 3;
```

```
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> SELECT * FROM t1;
```

```

+-----+-----+-----+
| col_a | col_b      | col_c |
+-----+-----+-----+
| 1      | Updated!   | 24    |
| 4      | second test| 43    |
| 3      | Updated!   | -2    |
| 4      | tenth test | 11    |
| 1      | Updated!   | 20    |
| 5      | third test | 100   |
+-----+-----+-----+
6 rows in set (0.00 sec)

```

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 5;
```

```
Query OK, 1 row affected (0.00 sec)
```

```
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> SELECT * from t1;
```

```

+-----+-----+-----+
| col_a | col_b      | col_c |
+-----+-----+-----+
| 1      | Updated!   | 24    |
| 4      | second test| 43    |
| 3      | Updated!   | -2    |
| 4      | tenth test | 11    |
| 1      | Updated!   | 20    |
| 5      | Updated!   | 100   |
+-----+-----+-----+
6 rows in set (0.02 sec)

```

```
mysql> DELETE FROM t1 WHERE col_a = 1;
```


Query OK, 2 rows affected (0.00 sec)

mysql> SELECT * FROM t1;

col_a	col_b	col_c
4	second test	43
3	Updated!	-2
4	tenth test	11
5	Updated!	100

4 rows in set (0.00 sec)

mysql> DELETE FROM t1 WHERE col_a = 3;

Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t1;

col_a	col_b	col_c
4	second test	43
4	tenth test	11
5	Updated!	100

3 rows in set (0.00 sec)

mysql> DELETE FROM t1 WHERE col_a = 5;

Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t1;

col_a	col_b	col_c
4	second test	43
4	tenth test	11

2 rows in set (0.00 sec)

mysql> RENAME TABLE t1 TO t2;

Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t2;

col_a	col_b	col_c
4	second test	43
4	tenth test	11

```
2 rows in set (0.00 sec)
```

```
mysql> DROP TABLE t2;
```

```
Query OK, 0 rows affected (1.69 sec)
```

Spartan存储引擎的第4阶段开发工作就此圆满结束。它现在已经是一个支持读/写/更新/删除操作的存储引擎了。在下一阶段，我们将为它添加一个索引类以提高查询工作的效率。

7.2.7 阶段 5：数据的索引

这个阶段的目标是在第4阶段的基础上为存储引擎添加索引支持。Spartan存储引擎的索引功能只支持每个表有一个索引，但你只要稍加努力就可以让它支持每个表有多个索引的情况。这个阶段将演示如何使用Spartan_index类为表建立和使用索引。因为需要修改的地方非常多，所以建议大家先把这一节的内容全部看完后再动手进行那些修改。

首先，需要把Spartan_index类的文件添加到项目文件里去。如果你使用的是Linux平台，将需要编辑你在第1阶段制作的Makefile.am和Makefile.in文件，用如下所示的命令，把Spartan_index类的头文件和类文件添加到适当的地方。

Makefile.am文件应该包含以下内容（在第377行附近）。

```
noinst_HEADERS      = ha_spartan.h spartan_data.h spartan_index.h
libspartan_a_SOURCES = ha_spartan.cc spartan_data.cc spartan_index.cc
```

Makefile.in文件应该包含以下内容（在第91行附近）。

```
noinst_HEADERS      = ha_spartan.h spartan_data.h spartan_index.h
libspartan_a_SOURCES = ha_spartan.cc spartan_data.cc spartan_index.cc
am_libspartan_a_OBJECTS = ha_spartan.$(OBJEXT) spartan_data.$(OBJEXT) \
    spartan_index.$(OBJEXT)
```

如果你使用的是Windows平台，把这两个文件添加到spartan项目里即可。

Spartan_index类将为Spartan_data类里的行保存一个与之对应的记录指针。当服务器使用主键搜索一条记录的时候，它将先使用Spartan_index类去查找相应的记录指针，再通过Spartan_index类发出一个“直接读”调用，把那条记录直接检索出来。与进行一次全表扫描的情况相比，使用索引将大大提高随机读取一条给定记录的速度。

本节中的源代码只实现了最基本的索引操作。虽然还要取决于你发出的查询命令有多么复杂，但我们将要进行的那些修改对绝大多数情况来说应该是足够的。我将详细介绍每一处需要修改的地方并在完成每处修改后给出有关函数的完整源代码。

1. 更新Spartan存储引擎的源文件

Spartan_index类做的事情很简单：把每条记录在数据文件里的位置和该记录的键保存起来。你需要在ha_spartan.cc文件里修改的方法包括：index_read()、index_read_idx()、index_next()、index_prev()、index_first()和index_last()。这些方法按用途可以分为三大类：从索引里读取索引数据、遍历整个索引、访问前一个和后一个索引项（对第一个和最后一个索引项的访问属于这种情况的特例）。我已经在Spartan_index类里把所有这些操作都实现出来了。

(1) 更新头文件

要想使用Spartan_index类，必须先在ha_spartan.h文件里添加一个对spartan_index.h文件的引用。代码清单7-34给出了完成修改后的源代码片段（为简明起见，我省略了注释）。在完成这个修改后，请重新编译spartan项目的源文件以确认没有任何错误。

代码清单7-34 修改ha_spartan.h文件里的st_spartan_share结构

```
typedef struct st_spartan_share {
    char *table_name;
    uint table_name_length, use_count;
    pthread_mutex_t mutex;
    THR_LOCK lock;
    Spartan_data *data_class;
    Spartan_index *index_class;
} SPARTAN_SHARE;
```

打开ha_spartan.h文件并添加一条#include指令来引入spartan_index.h头文件，然后在st_spartan_share结构里添加一个对象引用指针。代码清单7-35给出了完成修改后的源代码片段（为简明起见，我省略了注释）。在完成这个修改之后，请重新编译spartan项目的源文件以确认没有任何错误。

代码清单7-35 修改ha_spartan.h文件里的st_spartan_share结构

```
#include "spartan_data.h"
#include "spartan_index.h"
#ifdef USE_PRAGMA_INTERFACE
#pragma interface      /* gcc class implementation */
#endif
```

...

```
typedef struct st_spartan_share {
    char *table_name;
    uint table_name_length, use_count;
    pthread_mutex_t mutex;
    THR_LOCK lock;
    Spartan_data *data_class;
    Spartan_index *index_class;
} SPARTAN_SHARE;
```

ha_spartan.h文件里还有几处地方需要修改。需要添加一些操作标志来告诉优化器Spartan存储引擎都支持哪些索引操作。还需要为索引参数设置一些边界：最多支持多少个键、键的最大长度、键最多可以由几个部分构成等。就第5阶段而言，按照代码清单7-36给出的数值进行设置就可以了。我把需要在这个文件里修改的地方汇总在了这份清单里。请注意table_flags()方法，需要通过它去告诉优化器存储引擎都有那些限制。以Spartan存储引擎为例，我禁用了BLOB和自动递增字段。这些标志的完整清单可以在handler.h文件里查到。

代码清单7-36 修改ha_spartan.h文件里的ha_spartan类定义

```
const char *index_type(uint inx) { return "Spartan_index class"; }
const char **bas_ext() const;
```

```

...
ulong table_flags() const
{
    return (HA_NO_BLOBS | HA_NO_AUTO_INCREMENT);
}
...
ulong index_flags(uint inx, uint part, bool all_parts) const
{
    return (HA_READ_NEXT | HA_READ_PREV | HA_READ_RANGE |
            HA_READ_ORDER | HA_KEYREAD_ONLY);
}
...
uint max_supported_keys()          const { return 1; }
uint max_supported_key_parts()      const { return 1; }
uint max_supported_key_length()     const { return 128; }

```

除了上面这份清单里的修改，还需要添加一些东西。检索某给定记录的键将是一个使用频率相当高的操作，为了让事情变得容易些，我编写了两个辅助方法：一个是`get_key()`，用来查找键字段并返回该字段的值（如果没有键，返回0）；另一个是`get_key_len()`，用来返回键的长度。需要把这两个方法的定义也添加到Spartan存储引擎的类文件（`ha_spartan.h`）里去，如下所示。

```

byte *get_key();
int get_key_len();

```

读者将需要在`ha_spartan.cc`类文件里实现这些方法。

(2) 更新类文件

准备工作结束后，可以开始在Spartan存储引擎的类文件里实现各种索引方法了。打开`ha_spartan.cc`文件并找到`get_share()`方法，把Spartan_index类的实例化代码添加到`get_share()`方法里去。代码清单7-37是完成这个修改后的`get_share()`方法。

代码清单7-37 修改`ha_spartan.cc`文件里的`get_share()`方法

```

static SPARTAN_SHARE *get_share(const char *table_name, TABLE *table)
{
    SPARTAN_SHARE *share;
    uint length;
    char *tmp_name;

    pthread_mutex_lock(&spartan_mutex); //create a lock for single access
    length=(uint) strlen(table_name);

    if (!(share=(SPARTAN_SHARE*) hash_search(&spartan_open_tables,
                                              (byte*) table_name,
                                              length)))
    {
        /*
        Allocate several memory blocks at one time.
        Note: my_multi_malloc takes MySQL flags
        (set to zero fill and with extra error checking),
        one or more pairs of addresses and size of memory to allocate.

```



```

*/
if (!my_multi_malloc(MYF(MY_WME | MY_ZEROFILL),
                    &share, sizeof(*share),
                    &tmp_name, length+1,
                    NULLS))
{
    pthread_mutex_unlock(&spartan_mutex);
    return NULL;
}
/*
    Set the initial variables to defaults.
*/
share->use_count=0;
share->table_name_length=length;
share->table_name = (char *)my_malloc(length + 1, MYF(0));
strcpy(share->table_name, table_name);
/*
    Insert table name into hash for future reference.
*/
if (my_hash_insert(&spartan_open_tables, (byte*) share))
    goto error;
thr_lock_init(&share->lock);
/*
    Create an instance of data class
*/
share->data_class = new Spartan_data();
/*
    Create an instance of index class
*/
share->index_class = new Spartan_index();
pthread_mutex_init(&share->mutex, MY_MUTEX_INIT_FAST);
}
share->use_count++; // increment use count on reference
pthread_mutex_unlock(&spartan_mutex); //release mutex lock
return share;

error:
pthread_mutex_destroy(&share->mutex);
my_free((gptr) share, MYF(0));

return NULL;
pthread_mutex_init(&share->mutex, MY_MUTEX_INIT_FAST);
}
}

```

相应的，还需要在释放share结构的时候把这个对象引用指针也释放掉。请找到free_share()方法并把释放index_class对象引用指针的代码添加进去。代码清单7-38是完成这个修改后的free_share()方法。

代码清单7-38 修改ha_spartan.cc文件里的free_share()方法

```
static int free_share(SPARTAN_SHARE *share)
{
    DEBUG_ENTER("ha_spartan::free_share");
    pthread_mutex_lock(&spartan_mutex);
    if (--share->use_count)
    {
        if (share->data_class != NULL)
            delete share->data_class;
        share->data_class = NULL;
        if (share->index_class != NULL)
            delete share->index_class;
        share->index_class = NULL;
        hash_delete(&spartan_open_tables, (byte*) share);
        thr_lock_delete(&share->lock);
        pthread_mutex_destroy(&share->mutex);
        my_free((gptr)share->table_name, MYF(0));
    }
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}
```

在完成这些修改之后，重新编译spartan项目并查纠错误。接下来，还需要修改一些方法，让它们能够支持索引。

需要做的第一件事情是回到open()、create()、close()、write_row()、update_row()、delete_row()和rename_table()等方法，并添加对索引类的调用来维护索引。具体地说，就是在这些方法添加一些代码来查找键字段、保存键及其检索键的位置。

open()方法既要打开数据文件，也要打开索引文件。唯一的额外步骤是还要把索引加载到内存里。打开Spartan存储引擎的类文件(ha_spartan.cc)并找到open()方法，把用来打开索引文件并把索引加载到内存里去的Spartan_index方法调用添加进去。代码清单7-39给出了完成这些修改后的open()方法。

代码清单7-39 修改ha_spartan.cc文件里的open()方法

```
int ha_spartan::open(const char *name, int mode, uint test_if_locked)
{
    DEBUG_ENTER("ha_spartan::open");
    char name_buff[FN_REFLen];

    if (!(share = get_share(name, table)))
        DEBUG_RETURN(1);
    share->data_class->open_table(fn_format(name_buff, name, "", SDE_EXT,
                                           MY_REPLACE_EXT|MY_UNPACK_FILENAME));
    share->index_class->open_index(fn_format(name_buff, name, "", SDI_EXT,
                                           MY_REPLACE_EXT|MY_UNPACK_FILENAME));
    share->index_class->load_index();
    current_position = 0;
```

```

    thr_lock_data_init(&share->lock,&lock,NULL);
    DBUG_RETURN(0);
}

```

create()方法既要创建数据文件,也要创建索引文件。在Spartan存储引擎的类文件里找到create()方法,把用来创建索引文件的Spartan_index方法调用添加到索引类中。代码清单7-40给出了完成这些修改后的create()方法。

代码清单7-40 修改ha_spartan.cc文件里的create()方法

```

int ha_spartan::create(const char *name, TABLE *table_arg,
                      HA_CREATE_INFO *create_info)
{
    DBUG_ENTER("ha_spartan::create");
    char name_buff[FN_REFLen];
    if (!(share = get_share(name, table)))
        DBUG_RETURN(1);
    if (share->data_class->create_table(fn_format(name_buff, name, "", SDE_EXT,
                                                MY_REPLACE_EXT|MY_UNPACK_FILENAME)))
        DBUG_RETURN(-1);
    if (share->index_class->create_index(fn_format(name_buff, name, "", SDI_EXT,
                                                MY_REPLACE_EXT|MY_UNPACK_FILENAME),
                                        128))
        DBUG_RETURN(-1);
    share->index_class->close_index();
    share->data_class->close_table();
    DBUG_RETURN(0);
}

```

close()方法现在既要关闭数据文件,也要关闭索引文件。请注意, Spartan_index类使用了一个驻留在内存里的结构来保存索引数据及其变化情况,必须先把这个结构写入磁盘才能关闭索引文件。在Spartan存储引擎的类文件里找到close()方法,把用来保存、销毁那个驻留在内存里的索引结构和用来关闭索引文件的Spartan_index方法调用添加进去。代码清单7-41给出了完成这些修改后的close()方法。

代码清单7-41 修改ha_spartan.cc文件里的close()方法

```

int ha_spartan::close(void)
{
    DBUG_ENTER("ha_spartan::close");
    share->data_class->close_table();
    share->index_class->save_index();
    share->index_class->destroy_index();
    share->index_class->close_index();
    DBUG_RETURN(free_share(share));
}

```

接下来需要修改的是读、写方法。因为存在着表没有使用任何键的可能性,这个方法必须检查被读/写的记录有没添加键。为了让事情变得容易一些,我编写了两个辅助方法:一个是get_key(),用

来查找键字段并返回该字段的值（如果没有键，返回0）；另一个是get_key_len()，用来返回键的长度。代码清单7-42给出了这两个辅助方法的源代码，请把它们添加到ha_spartan.cc文件里去。

代码清单7-42 给ha_spartan.cc文件添加两个辅助方法

```
byte *ha_spartan::get_key()
{
    byte *key = 0;

    DEBUG_ENTER("ha_spartan::get_key");
    /*
     * For each field in the table, check to see if it is the key
     * by checking the key_start variable. (1 = is a key).
     */
    for (Field **field=table->field ; *field ; field++)
    {
        if ((*field)->key_start.to_ulonglong() == 1)
        {
            /*
             * Copy field value to key value (save key)
             */
            key = (byte *)my_malloc((*field)->field_length,
                                   MYF(MY_ZEROFILL | MY_WME));
            memcpy(key, (*field)->ptr, (*field)->key_length());
        }
    }
    DEBUG_RETURN(key);
}

int ha_spartan::get_key_len()
{
    int length = 0;

    DEBUG_ENTER("ha_spartan::get_key");
    /*
     * For each field in the table, check to see if it is the key
     * by checking the key_start variable. (1 = is a key).
     */
    for (Field **field=table->field ; *field ; field++)
    {
        if ((*field)->key_start.to_ulonglong() == 1)
        {
            /*
             * Copy field length to key length
             */
            length = (*field)->key_length();
        }
    }
    DEBUG_RETURN(length);
}
```


write_row()方法现在需要在把一条记录写入其数据文件的同时把该记录的键插入其索引文件。在Spartan存储引擎的类文件里找到write_row()方法，将用来把一个键（如果该记录有键的话）插入索引文件的Spartan_index方法调用添加进去。代码清单7-43给出了完成这些修改后的write_row()方法。

代码清单7-43 修改ha_spartan.cc文件里的write_row()方法

```
int ha_spartan::write_row(byte * buf)
{
    long long pos;
    SDE_INDEX ndx;

    DBUG_ENTER("ha_spartan::write_row");
    ha_statistic_increment(&SSV::ha_write_count);
    ndx.length = get_key_len();
    memcpy(ndx.key, get_key(), get_key_len());
    pthread_mutex_lock(&spartan_mutex);
    pos = share->data_class->write_row(buf, table->s->rec_buff_length)
    ndx.pos = pos;
    if (ndx.key != 0)
        share->index_class->insert_key(&ndx, false);
    pthread_mutex_unlock(&spartan_mutex);
    DBUG_RETURN(0);
}
```

update_row()方法现在需要修改数据文件里的一条记录和索引文件里的一个键。因为Spartan_index类使用了一个驻留在内存里的结构来保存索引数据及其变化情况，对索引文件的修改将需要三个步骤：修改键、把它写入磁盘、重新加载它。

注解 有经验的程序员会注意到，只要对Spartan_index类的源代码做一些小修改，就可以省略重新加载索引文件的步骤。你知道是什么吗？我来提示一下：如果Spartan_index类的update_key()方法先修改键，再把它重新放到内存结构里会怎样？把这个问题留给你作为练习。欢迎大家研究和改进Spartan_index类的代码。

在Spartan存储引擎的类文件里找到update_row()方法，把用来修改一个键（如果该记录有键的话）的Spartan_index方法调用添加进去。代码清单7-44给出了完成这些修改后的update_row()方法。

代码清单7-44 修改ha_spartan.cc文件里的update_row()方法

```
int ha_spartan::update_row(const byte * old_data, byte * new_data)
{
    DBUG_ENTER("ha_spartan::update_row");
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->update_row((byte *)old_data, new_data,
        table->s->rec_buff_length, current_position -
        share->data_class->row_size(table->s->rec_buff_length));
}
```

```

if (get_key() != 0)
{
    share->index_class->update_key(get_key(), current_position -
                                   share->data_class->row_size(table->s->rec_buff_length),
                                   get_key_len());
    share->index_class->save_index();
    share->index_class->load_index();
}
pthread_mutex_unlock(&spartan_mutex);
DEBUG_RETURN(0);
}

```

delete_row()方法不那么复杂。具体到这个例子里，它只要删除数据文件里的记录，并从那个驻留在内存里的索引结构里删除该记录的键（如果该记录有键的话）就行了。在Spartan存储引擎的类文件里找到delete_row()方法，把用来删除一个键（如果该记录有键的话）的方法调用添加进去。代码清单7-45给出了完成这些修改后的delete_row()方法。

代码清单7-45 修改ha_spartan.cc文件里的delete_row()方法

```

int ha_spartan::delete_row(const byte * buf)
{
    long long pos;

    DEBUG_ENTER("ha_spartan::delete_row");
    if (current_position > 0)
        pos = current_position -
              share->data_class->row_size(table->s->rec_buff_length);
    else
        pos = 0;
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->delete_row((byte *)buf,
                                table->s->rec_buff_length, pos);
    if (get_key() != 0)
        share->index_class->delete_key(get_key(), pos, get_key_len());
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}

```

对用来删除所有行的delete_all_rows()方法也需要做类似的修改。在Spartan存储引擎的类文件里找到delete_all_row()方法，把用来删除全部索引和截短索引文件的Spartan_index方法调用添加进去。代码清单7-46给出了完成这些修改后的delete_all_row()方法。

代码清单7-46 修改ha_spartan.cc文件里的delete_all_row()方法

```

int ha_spartan::delete_all_rows()
{
    DEBUG_ENTER("ha_spartan::delete_all_rows");
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->trunc_table();
    share->index_class->destroy_index();
}

```

```

share->index_class->trunc_index();
pthread_mutex_unlock(&spartan_mutex);
DEBUG_RETURN(0);
}

```

delete_table()方法现在需要删除数据文件和索引文件。在Spartan存储引擎的类文件里找到delete_table()方法,把用来销毁驻留在内存里的索引结构、关闭索引文件以及调用my_delete()函数删除索引文件的代码添加进去。代码清单7-47给出了完成这些修改后的delete_table()方法。

代码清单7-47 修改ha_spartan.cc文件里的delete_table()方法

```

int ha_spartan::delete_table(const char *name)
{
    DEBUG_ENTER("ha_spartan::delete_table");
    char name_buff[FN_REFLLEN];

    if (!(share = get_share(name, table)))
        DEBUG_RETURN(1);
    pthread_mutex_lock(&spartan_mutex);
    share->data_class->close_table();
    /*
     * Destroy the index in memory and close it.
     */
    share->index_class->destroy_index();
    share->index_class->close_index();
    /*
     * Call the mysql delete file method.
     * Note: the fn_format() method correctly creates a file name from the
     * name passed into the method.
     */
    my_delete(fn_format(name_buff, name, "", SDE_EXT,
        MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
    /*
     * Call the mysql delete file method.
     * Note: the fn_format() method correctly creates a file name from the
     * name passed into the method.
     */
    my_delete(fn_format(name_buff, name, "", SDI_EXT,
        MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
    pthread_mutex_unlock(&spartan_mutex);
    DEBUG_RETURN(0);
}

```

最后一个需要修改的通用文件读写操作是rename_table()方法。对这个方法的修改套路与前面那些修改大同小异。在Spartan存储引擎的类文件里找到rename_table()方法,把用来复制索引文件的代码添加进去。代码清单7-48给出了完成这些修改后的rename_table()方法。

代码清单7-48 修改ha_spartan.cc文件里的rename_table()方法

```

int ha_spartan::rename_table(const char * from, const char * to)
{

```

```

DEBUG_ENTER("ha_spartan::rename_table ");
char data_from[FN_REFLen];
char data_to[FN_REFLen];
char index_from[FN_REFLen];
char index_to[FN_REFLen];

if (!(share = get_share(from, table)))
    DEBUG_RETURN(1);
/*
    Begin critical section by locking the spartan mutex variable.
*/
pthread_mutex_lock(&spartan_mutex);
/*
    Close the table then copy it then reopen new file.
*/
share->data_class->close_table();
my_copy(fn_format(data_from, from, "", SDE_EXT,
    MY_REPLACE_EXT|MY_UNPACK_FILENAME),
    fn_format(data_to, to, "", SDE_EXT,
    MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
share->data_class->open_table(data_to);
share->index_class->close_index();
my_copy(fn_format(index_from, from, "", SDI_EXT,
    MY_REPLACE_EXT|MY_UNPACK_FILENAME),
    fn_format(index_to, to, "", SDI_EXT,
    MY_REPLACE_EXT|MY_UNPACK_FILENAME), MYF(0));
share->index_class->open_index(index_to);
/*
    End critical section by unlocking the spartan mutex variable.
*/
pthread_mutex_unlock(&spartan_mutex);
/*
    Delete the file using MySQL's delete file method.
*/
my_delete(data_from, MYF(0));
my_delete(index_from, MYF(0));
DEBUG_RETURN(0);
}

```

终于改完了！需要修改的地方还真不少。正如你们看到的那样，添加索引支持让有关的代码变得复杂了许多。希望这些修改能让你体会到MySQL自带的那些存储引擎有多么了不起。接下来，将开始对具体完成各种索引操作的方法进行修改。

要想为一个阶段5存储引擎添加索引机制，有6个方法是必须实现的。在这些方法当中，有几个将根据给定的索引键从数据文件返回一个记录，另外几个则返回一个键。MySQL的随机文档在这方面说得不是很细致，从那些参数的名字上也看不出什么眉目，但我将演示如何使用它们。这些方法必须返回一个“键未找到”或“已到达文件尾”的返回代码。一定要把这些返回语句的代码安排好，否则就有可能遇到一些奇怪的查询结果。

需要实现的第一个索引方法是index_read()方法。这个方法将根据给定的键从索引文件里把相应的记录读入行缓冲区。如果你传递给它的键是空值，它应该返回文件里的第一个键值。在Spartan存储引擎的类文件里找到index_read()方法，把从索引获得一个文件读写位置、再从数据文件里读出相应记录的代码添加进去。代码清单7-49给出了完成这些修改后的index_read()方法。

代码清单7-49 修改ha_spartan.cc文件里的index_read()方法

```
int ha_spartan::index_read(byte * buf, const byte * key,
                           uint key_len __attribute__((unused)),
                           enum ha_rkey_function find_flag
                           __attribute__((unused)))
{
    long long pos;

    DEBUG_ENTER("ha_spartan::index_read");
    if (key == NULL)
        pos = share->index_class->get_first_pos();
    else
        pos = share->index_class->get_index_pos((byte *)key, key_len);
    if (pos == -1)
        DEBUG_RETURN(HA_ERR_KEY_NOT_FOUND);
    current_position =
        pos + share->data_class->row_size(table->s->rec_buff_length);
    share->data_class->read_row(buf, table->s->rec_buff_length, pos);
    share->index_class->get_next_key();
    DEBUG_RETURN(0);
}
```

下一个索引方法是index_read_idx()。它与index_read()方法很相似，只不过它是从优化器的其他部分被调用的（比如当最多只有一个匹配的时候——详见sql_select.cc文件）。这个方法将根据给定的键从索引文件里把相应的记录读入行缓冲区。如果传递给它的键是空值，它应该返回文件里的第一个键值和第一个行。在Spartan存储引擎的类文件里找到index_read_idx()方法，把从索引获得一个文件读写位置、再从数据文件里读出相应记录的代码添加进去。代码清单7-50给出了完成这些修改后的index_read_idx()方法。

代码清单7-50 修改ha_spartan.cc文件里的index_read_idx()方法

```
int ha_spartan::index_read_idx(byte * buf, uint index, const byte * key,
                                uint key_len __attribute__((unused)),
                                enum ha_rkey_function find_flag
                                __attribute__((unused)))
{
    long long pos;

    DEBUG_ENTER("ha_spartan::index_read_idx");
    pos = share->index_class->get_index_pos((byte *)key, key_len);
    if (pos == -1)
        DEBUG_RETURN(HA_ERR_KEY_NOT_FOUND);
}
```

```

share->data_class->read_row(buf, table->s->rec_buff_length, pos);
DEBUG_RETURN(0);
}

```

下一个索引方法是`index_next()`。这个方法将从索引里查出下一个键并从数据文件返回与该键匹配的行。这称为区间索引扫描（**range index scan**）。在Spartan存储引擎的类文件里找到`index_next()`方法，添加代码来从索引获得下一个键，再从数据文件里读出一行数据。代码清单7-51给出了完成这些修改后的`index_next()`方法。

代码清单7-51 修改`ha_spartan.cc`文件里的`index_next()`方法

```

int ha_spartan::index_next(byte * buf)
{
    byte *key = 0;
    long long pos;

    DEBUG_ENTER("ha_spartan::index_next");
    key = share->index_class->get_next_key();
    if (key == 0)
        DEBUG_RETURN(HA_ERR_END_OF_FILE);
    pos = share->index_class->get_index_pos((byte *)key, get_key_len());
    share->index_class->seek_index(key, get_key_len());
    share->index_class->get_next_key();
    if (pos == -1)
        DEBUG_RETURN(HA_ERR_KEY_NOT_FOUND);
    share->data_class->read_row(buf, table->s->rec_buff_length, pos);
    DEBUG_RETURN(0);
}

```

下一个索引方法也是一个用来完成区间查询（**range query**）的方法。`index_prev()`方法将从索引里查出上一个键并从数据文件返回与该键匹配的行。它将在区间索引扫描期间被调用。在Spartan存储引擎的类文件里找到`index_prev()`方法，把从索引获得下一个键、再从数据文件里读出相应记录的代码添加进去。代码清单7-52给出了完成这些修改后的`index_prev()`方法。

代码清单7-52 修改`ha_spartan.cc`文件里的`index_prev()`方法

```

int ha_spartan::index_prev(byte * buf)
{
    byte *key = 0;
    long long pos;

    DEBUG_ENTER("ha_spartan::index_prev");
    key = share->index_class->get_prev_key();
    if (key == 0)
        DEBUG_RETURN(HA_ERR_END_OF_FILE);
    pos = share->index_class->get_index_pos((byte *)key, get_key_len());
    share->index_class->seek_index(key, get_key_len());
    share->index_class->get_prev_key();
    if (pos == -1)

```

```

        DEBUG_RETURN(HA_ERR_KEY_NOT_FOUND);
    share->data_class->read_row(buf, table->s->rec_buff_length, pos);
    DEBUG_RETURN(0);
}

```

请注意，为了让`index_next()`和`index_prev()`方法的代码能够正确执行，我对索引指针做了一点儿小调整。区间查询在它第一次被使用时将生成两个对索引类的调用。第一个用来获得第1个键(`index_read`)，第二个用来获得下一个键(`index_next`)，以后的索引调用将只调用`index_next()`方法。因此，必须调用Spartan_index类的`get_prev_key()`方法才能正确地对键进行复位。如果你想让索引类能够与MySQL里的区间查询更好地配合工作，这又将是一个对它进行改进的好机会。

下一个索引方法也是一个用来完成区间查询的方法。`index_first()`方法将从索引里查出第一个键并返回。在Spartan存储引擎的类文件里找到`index_first()`方法，把从索引获得第一个键并返回该键的代码添加进去。代码清单7-53给出了完成这些修改后的`index_first()`方法。

代码清单7-53 修改ha_spartan.cc文件里的`index_first()`方法

```

int ha_spartan::index_first(byte * buf)
{
    byte *key = 0;

    DEBUG_ENTER("ha_spartan::index_first");
    key = share->index_class->get_first_key();
    if (key == 0)
        DEBUG_RETURN(HA_ERR_END_OF_FILE);
    memcpy(buf, key, get_key_len());
    DEBUG_RETURN(0);
}

```

最后一个索引方法还是一个用来完成区间查询的方法。`index_last()`方法将从索引里查出最后一个键并返回。在Spartan存储引擎的类文件里找到`index_last()`方法，把从索引获得第一个键并返回该键的代码添加进去。代码清单7-54给出了完成这些修改后的`index_last()`方法。

代码清单7-54 修改ha_spartan.cc文件里的`index_last()`方法

```

int ha_spartan::index_last(byte * buf)
{
    byte *key = 0;

    DEBUG_ENTER("ha_spartan::index_last");
    key = share->index_class->get_last_key();
    if (key == 0)
        DEBUG_RETURN(HA_ERR_END_OF_FILE);
    memcpy(buf, key, get_key_len());
    DEBUG_RETURN(0);
}

```

好了，现在可以编译MySQL服务器并调试其中的错误了。在编译工作结束后，你将得到一个阶段5存储引擎。剩下的事情就是编译MySQL服务器和对它进行测试了。

2. 测试Spartan存储引擎（阶段5）

当再次运行spartandb测试的时候，应该看到所有的语句都执行成功了。在开始测试索引操作之前，应该先确认阶段4引擎里的一切仍都工作正常。

索引测试需要提前创建一个表并在其中填充一些数据。可以像以前那样用普通的INSERT语句来填充数据，现在你需要测试的东西是索引。请输入一条如下所示的SELECT命令，在它的WHERE子句里要用上你的索引列（col_a）：

```
SELECT * FROM t1 WHERE col_a = 2;
```

在执行这条命令之后，你应该看到它返回的行。但这证明不了什么——Spartan存储引擎早就可以返回这样的行了。如果你想知道索引有没有起作用，最好的办法是用一个索引值的变化范围很大的大表来进行测试。要创建一个这样的表当然要花一些时间，但我建议你去这样做。

还有另一个办法。你可以启动MySQL服务器并用你的调试器在源代码里设置一些断点，然后发出一些会用到索引的查询命令。这听起来很麻烦，你可能也没有足够的时间而只能试几个例子。这不是什么大问题，因为可以把那些测试添加到spartan测试文件里去。你可以在CREATE TABLE命令里添加一个键列，再多添加几条用来进行点查询和区间查询的SELECT ... WHERE命令。代码清单7-55给出了修改后的spartandb.test文件。

代码清单7-55 Spartan存储引擎的测试文件（spartandb.test）

```
#
# Simple test for the Spartan storage engine
#
--disable_warnings
drop table if exists t1;
--enable_warnings

CREATE TABLE t1 (
  col_a int KEY,
  col_b varchar(20),
  col_c int
) ENGINE=SPARTAN;

INSERT INTO t1 VALUES (1, "first test", 24);
INSERT INTO t1 VALUES (2, "second test", 43);
INSERT INTO t1 VALUES (9, "fourth test", -2);
INSERT INTO t1 VALUES (3, 'eighth test', -22);
INSERT INTO t1 VALUES (4, "tenth test", 11);
INSERT INTO t1 VALUES (8, "seventh test", 20);
INSERT INTO t1 VALUES (5, "third test", 100);
SELECT * FROM t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 1;
SELECT * from t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 3;
SELECT * from t1;
UPDATE t1 SET col_b = "Updated!" WHERE col_a = 5;
SELECT * from t1;
DELETE FROM t1 WHERE col_a = 1;
```



```
SELECT * FROM t1;
DELETE FROM t1 WHERE col_a = 3;
SELECT * FROM t1;
DELETE FROM t1 WHERE col_a = 5;
SELECT * FROM t1;
SELECT * FROM t1 WHERE col_a = 4;
SELECT * FROM t1 WHERE col_a >= 2 AND col_a <= 5;
SELECT * FROM t1 WHERE col_a = 22;
DELETE FROM t1 WHERE col_a = 5;
SELECT * FROM t1;
SELECT * FROM t1 WHERE col_a = 5;
UPDATE t1 SET col_a = 99 WHERE col_a = 8;
SELECT * FROM t1 WHERE col_a = 8;
SELECT * FROM t1 WHERE col_a = 99;
RENAME TABLE t1 TO t2;
SELECT * FROM t2;
DROP TABLE t2;
```

请注意，我修改了一些INSERT语句以便让索引方法能够起作用。运行这个测试并查看其结果，代码清单7-56是这个测试的一个预期结果示例。当你用MySQL Test Suite工具运行这个测试的时候，它应该不出任何错误地完成测试。

代码清单7-56 Spartan存储引擎的测试结果示例（阶段5）

```
mysql> CREATE TABLE t1 (
->   col_a int KEY,
->   col_b varchar(20),
->   col_c int
-> ) ENGINE=SPARTAN;
Query OK, 0 rows affected (0.02 sec)

mysql>
mysql> SELECT * FROM t1;
Empty set (0.02 sec)

mysql> INSERT INTO t1 VALUES(1, "first test", 24);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES(2, "second test", 43);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES(9, "fourth test", -2);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t1 VALUES (3, 'eighth test', -22);
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO t1 VALUES(4, "tenth test", 11);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO t1 VALUES(8, "seventh test", 20);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO t1 VALUES(5, "third test", 100);
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT * FROM t1;
```

col_a	col_b	col_c
1	first test	24
2	second test	43
9	fourth test	-2
3	eighth test	-22
4	tenth test	11
8	seventh test	20
5	third test	100

```
7 rows in set (0.00 sec)
```

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> SELECT * from t1;
```

col_a	col_b	col_c
1	Updated!	24
2	second test	43
9	fourth test	-2
3	eighth test	-22
4	tenth test	11
8	seventh test	20
5	third test	100

```
7 rows in set (0.00 sec)
```

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 3;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

```
mysql> SELECT * from t1;
```

col_a	col_b	col_c
1	Updated!	24
2	second test	43
9	fourth test	-2
3	Updated!	-22

4	tenth test	11
8	seventh test	20
5	third test	100

```
+-----+-----+-----+
```

7 rows in set (0.00 sec)

```
mysql> UPDATE t1 SET col_b = "Updated!" WHERE col_a = 5;
```

Query OK, 0 rows affected (0.00 sec)

Rows matched: 0 Changed: 0 Warnings: 0

```
mysql> SELECT * from t1;
```

```
+-----+-----+-----+
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

1	Updated!	24
2	second test	43
9	fourth test	-2
3	Updated!	-22
4	tenth test	11
8	seventh test	20
5	Updated!	100

```
+-----+-----+-----+
```

7 rows in set (0.00 sec)

```
mysql> DELETE FROM t1 WHERE col_a = 1;
```

Query OK, 1 row affected (0.00 sec)

```
mysql> SELECT * FROM t1;
```

```
+-----+-----+-----+
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

2	second test	43
9	fourth test	-2
3	Updated!	-22
4	tenth test	11
8	seventh test	20
5	Updated!	100

```
+-----+-----+-----+
```

6 rows in set (0.00 sec)

```
mysql> DELETE FROM t1 WHERE col_a = 3;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> SELECT * FROM t1;
```

```
+-----+-----+-----+
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

2	second test	43
9	fourth test	-2
4	tenth test	11

8	seventh test	20
5	Updated!	100

```
+-----+-----+-----+
```

5 rows in set (0.00 sec)

```
mysql> DELETE FROM t1 WHERE col_a = 5;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> SELECT * FROM t1;
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

2	second test	43
---	-------------	----

9	fourth test	-2
---	-------------	----

4	tenth test	11
---	------------	----

8	seventh test	20
---	--------------	----

```
+-----+-----+-----+
```

4 rows in set (0.00 sec)

```
mysql> SELECT * FROM t1 WHERE col_a = 4;
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

4	tenth test	11
---	------------	----

```
+-----+-----+-----+
```

1 row in set (0.00 sec)

```
mysql> SELECT * FROM t1 WHERE col_a >= 2 AND col_a <= 5;
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

2	second test	43
---	-------------	----

4	tenth test	11
---	------------	----

```
+-----+-----+-----+
```

2 rows in set (0.02 sec)

```
mysql> SELECT * FROM t1 WHERE col_a = 22;
```

Empty set (0.00 sec)

```
mysql> DELETE FROM t1 WHERE col_a = 5;
```

Query OK, 0 rows affected (0.00 sec)

```
mysql> SELECT * FROM t1;
```

col_a	col_b	col_c
-------	-------	-------

```
+-----+-----+-----+
```

2	second test	43
---	-------------	----

9	fourth test	-2
---	-------------	----

4	tenth test	11
---	------------	----

8	seventh test	20
---	--------------	----

```
+-----+-----+-----+
```



```

4 rows in set (0.00 sec)
mysql> SELECT * FROM t1 WHERE col_a = 5;
Empty set (0.00 sec)

mysql> UPDATE t1 SET col_a = 99 WHERE col_a = 8;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0

mysql> SELECT * FROM t1 WHERE col_a = 8;
Empty set (0.00 sec)

mysql> SELECT * FROM t1 WHERE col_a = 99;
+-----+-----+-----+
| col_a | col_b          | col_c |
+-----+-----+-----+
| 99    | seventh test   | 20    |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> RENAME TABLE t1 TO t2;
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM t2;
+-----+-----+-----+
| col_a | col_b          | col_c |
+-----+-----+-----+
| 2     | second test    | 43    |
| 9     | fourth test    | -2    |
| 4     | tenth test     | 11    |
| 99    | seventh test   | 20    |
+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> DROP TABLE t2;
Query OK, 0 rows affected (0.02 sec)

```

Spartan存储引擎的第5阶段开发工作就此圆满结束。它现在已经是一个能够利用索引来完成读/写/更新/删除操作的存储引擎了。MySQL自带的绝大多数存储引擎也不过实现到这个阶段而已。事实上，除了需要支持事务处理的环境，现在的Spartan存储引擎应该能够满足用户的存储需求。下一阶段将讨论一个更加复杂的问题：如何为存储引擎添加事务支持。

7.2.8 阶段 6：添加事务支持

多年以来，MySQL自带的存储引擎当中只有两个支持事务处理：BDB和InnoDB^①。事务机制允许一组操作作为一个原子操作来执行。比如说，在一个为银行开发的数据库系统里，从一个账户向另一

① 群集存储引擎（NDB）也支持事务。

个账户汇款的一系列操作应该一次完成，其间不允许有任何中断。如果那个数据库系统支持事务处理，我们就可以把这些操作封装为一个原子操作，如果在全部操作完成之前发生了意外，已经进行的修改将全部撤销，这样就不会发生钱（数据）从一个账户（表）取出，但永远到不了另一个账户（表）的事情了。代码清单7-57给出了一组用来完成这种银行转账操作的SQL语句，第一行和最后一条行语句是事务命令。

代码清单7-57 事务SQL命令示例

```
START TRANSACTION;
UPDATE SavingsAccount SET Balance = Balance-100
WHERE AccountNum = 123;
UPDATE CheckingAccount SET Balance = Balance + 100
WHERE AccountNum = 345;
COMMIT;
```

在实际工作中，如果需要进行快速访问，大部分数据库专家会选择MyISAM表类型；如果需要事务支持，大部分数据库专家会选择InnoDB。让人高兴的是，MySQL AB公司还为插件式存储引擎提供了支持事务的能力。

存储引擎里的事务支持需要通过start_stmt()和external_lock()方法来提供。调用start_stmt()方法将开始一次事务。external_lock()方法用来向表发出一个锁定信号，如果想明确地锁定一个表，就要发出这个调用。你的存储引擎必须在start_stmt()方法里实现新的事务，具体做法是创建一个保存点（savepoint），并调用trans_register_ha()方法向服务器注册这个事务。trans_register_ha()方法需要传递3个参数：当前线程、是否想让事务影响所有的线程、handler结构的地址。调用这个方法将开始一次事务。代码清单7-58给出了start_stmt()方法的一种实现。

代码清单7-58 start_stmt()方法的一种实现

```
int my_handler::start_stmt(THD *thd, thr_lock_type lock_type)
{
    DEBUG_ENTER("my_handler::index_last");
    int error= 0;
    /*
     * Save the transaction data
     */
    my_txn *txn= (my_txn *) thd->ha_data[my_handler_hon.slot];
    /*
     * If this is a new transaction, create it and save it to the
     * handler's slot in the ha_data array.
     */
    if (txn == NULL)
        thd->ha_data[my_handler_hon.slot]= txn= new my_txn;
    /*
     * Start the transaction and create a savepoint then register
     * the transaction.
     */
    if (txn->stmt == NULL && !(error= txn->tx_begin()))
```

```

{
    txn->stmt= txn->new_savepoint();
    trans_register_ha(thd, FALSE, &my_handler_hon);
}
DEBUG_RETURN(error);
}

```

从external_lock()方法开始一次事务稍微有点儿复杂。MySQL会在开始一次事务的时候，对该事务将会用到的每一个表发出一个external_lock()调用。因此，你必须在这个方法里通过检查trx->active_trans标志来判断是否有事务发生并对它做出适当的处理。为第一个表发出external_lock()调用也会隐含地开始一次事务操作。代码清单7-59给出了external_lock()方法的一种实现（有删节），可以在ha_innodb.cc文件里找到它的全部代码。

代码清单7-59 external_lock()方法的一种实现（选自InnoDB存储引擎的源代码）

```

int ha_innobase::external_lock(THD* thd, int Lock_type)
{
    row_prebuilt_t* prebuilt = (row_prebuilt_t*) innobase_prebuilt;
    trx_t* trx;

    DEBUG_ENTER("ha_innobase::external_lock");
    DEBUG_PRINT("enter", ("lock_type: %d", lock_type));

    update_thd(thd);

    trx = prebuilt->trx;

    prebuilt->sql_stat_start = TRUE;
    prebuilt->hint_need_to_fetch_extra_cols = 0;

    prebuilt->read_just_key = 0;
    prebuilt->keep_other_fields_on_keyread = FALSE;

    if (lock_type == F_WRLCK) {

        /* If this is a SELECT, then it is in UPDATE TABLE ...
        or SELECT ... FOR UPDATE */
        prebuilt->select_lock_type = LOCK_X;
        prebuilt->stored_select_lock_type = LOCK_X;
    }

    if (lock_type != F_UNLCK)
    {
        /* MySQL is setting a new table lock */

        trx->detailed_error[0] = '\0';

        /* Set the MySQL flag to mark that there is an active
        transaction */
        if (trx->active_trans == 0) {

```

```

        innobase_register_trx_and_stmt(thd);
        trx->active_trans = 1;
    } else if (trx->n_mysql_tables_in_use == 0) {
        innobase_register_stmt(thd);
    }

    trx->n_mysql_tables_in_use++;
    prebuilt->mysql_has_locked = TRUE;

...
    DEBUG_RETURN(0);
}

/* MySQL is releasing a table lock */

    trx->n_mysql_tables_in_use--;
    prebuilt->mysql_has_locked = FALSE;

    /* If the MySQL lock count drops to zero we know that the current SQL
    statement has ended */
    if (trx->n_mysql_tables_in_use == 0) {

...
        DEBUG_RETURN(0);
    }
}

```

知道了如何开始事务，我们再来看看如何停止它们，这又分为“提交”(commit)和“撤销”(roll back)两种情况。提交事务的意思是把此前的修改写入磁盘文件，保存相应的键并释放本次事务所占用的资源。MySQL AB公司在`handler_ton`结构里提供了一个方法(`int (*commit)(THD *thd, bool all)`)，可以用如下所示的函数描述来实现它。这个函数需要你传递两个参数：当前线程和是否想执行所有命令。

```
int (*commit)(THD *thd, bool all);
```

撤销一个事务就比较复杂了。此时，你必须撤销本次事务开始以来做过的所有事情。MySQL在`handler_ton`结构里提供了一个方法(`int (*rollback)(THD *thd, bool all)`)，可以用如下所示的函数描述来实现它。这个函数需要你传递两个参数：当前线程、你是否想撤销本次事务里的所有命令。

```
int (*rollback)(THD *thd, bool all);
```

为了实现事务，存储引擎必须提供某种缓存机制来存放还没有被存入数据库的改动。有些存储引擎使用某种堆结构；其他一些使用队列或类似的内部内存结构。如果想在存储引擎里实现事务，需要创建一种内部缓存机制并实现以下行为：如果事务最终被提交了，数据将从缓冲区里取出并写入磁盘文件；如果事务最终被撤销了，此前的操作都必须取消，并把它们做出的修改恢复原样。

保存点是另一种可以用来在事务期间管理有关数据的事务机制。保存点是一些允许你把信息存放在其中的内存区域，你可以使用它们来保存在事务过程中产生的数据。比如说，可以实现一个专用的

内部缓冲区来保存那些尚未被提交的数据。事实上，保存点的概念就是为了这种用途而形成的。

MySQL AB公司提供了好几种允许你在`handler_ton`结构里自行定义的保存点操作。这些操作出现在代码清单7-1中的`handler_ton`结构的第13到第15行。下面是这些保存点方法的方法描述。

```
uint savepoint_offset;
int (*savepoint_set)(THD *thd, void *sv);
int (*savepoint_rollback)(THD *thd, void *sv);
int (*savepoint_release)(THD *thd, void *sv);
```

`savepoint_offset`值是你打算保存的内存区域的长度。`savepoint_set()`方法允许你传一个值传递给参数`sv`，并把它保存为一个保存点。对`savepoint_rollback()`方法的调用发生在一个事务撤销操作被触发的时候，此时，服务器会把保存点`sv`里的信息返回给这个方法。类似的，对`savepoint_release()`方法的调用发生在MySQL服务器响应一个释放保存点事件的时候，服务器也将通过`sv`参数把被设置为保存点的数据返回给这个方法。关于保存点的更多信息请参见MySQL的源代码和MySQL的在线参考手册。

提示 如果你想知道事务机制的工作情况，在`ha_innodb.cc`和`ha_berkeley.cc`文件里可以找到许多经典的例子。在MySQL的在线参考手册上，也可以查到很多这方面的信息。

通过上述MySQL机制添加事务支持并不是故事的结束。使用了索引的存储引擎^①，必须提供一些允许事务的机制。这些操作必须能够把结点标记为“已被事务里的操作改变”，把修改前的原始数据保存起来直到事务结束为止。此时，所有的修改（索引和数据）将被提交给物理存储。这将需要修改`Spartan_index`类。

显然，在一个插件式存储引擎里实现事务，需要经过审慎的思考和计划。如果读者想在自己的存储引擎里实现事务支持，强烈建议你花些时间去研究BDB和InnoDB存储引擎，以及MySQL的在线参考手册。甚至可以启动调试器去观察建议的执行情况。不管选择哪种办法来实现事务支持，只要你能把它真地实现出来，它就肯定是一个独一无二的东西。只有少数几种优秀的存储引擎支持事务，但到目前为止，没有一种存储引擎能在这方面超越MySQL自带的存储引擎。

7.3 小结

本章介绍了插件式存储引擎的源代码并向大家演示了如何创建自己的存储引擎。通过`Spartan`存储引擎，你可以学到如何创建一个能够读写数据并支持并发访问和索引的存储引擎。虽然我对创建这个存储引擎的所有阶段都进行了解释，但我把添加事务支持的工作留给读者作为练习。

我没有为`Spartan`存储引擎实现所有的功能，而只实现了几个最基本的操作。既然你已经见识过这些基本操作的实现办法并有过这样一次经历，建议你在设计自己的存储引擎时再好好研究一下MySQL在线文档和源代码。

你也许会认为本章内容颇有难度，这很正常——为数据库系统创建一种物理存储机制可不是什么小项目。希望通过本章的学习能够加深对存储引擎创建过程的了解，并意识到创建那些实现索引和事

① 请注意，一个阶段6存储引擎可以不支持索引。索引对事务处理来说并不是必不可少的条件。不过，如果没有索引，事务处理的性能可能会受到很大的影响。

务支持的MySQL存储引擎是颇为不易的。这些工作没有一样是能够容易完成的。

最后，必须告诉大家，本章提供的数据类（Spartan_data类）和索引类（Spartan_index类）都有不少值得改进的地方。Spartan_data类对绝大多数应用项目来说可能还够用，但Spartan_index类的改进余地还是很大。如果你想以这两个类作为开发存储引擎的跳板，建议你先利用这两个类让自己的存储引擎运转起来，再考虑升级或替换它们的问题。

Spartan_index类有几个地方是我认为特别需要改进的。首先，它的内部缓冲区可以改进为一种更高效的树结构。这方面的选择有很多，B树或散列表机制都值得考虑。另一个值得改进的地方是Spartan_index类在处理区间查询（range query）时的做法。最后，如果你想为Spartan_index类添加事务支持，就还需要修改几个地方——你必须让这个类支持为了处理事务提交/撤销操作而选用的缓冲机制。

下一章将介绍一种比较流行的对MySQL系统进行扩展的办法。这包括添加自己的用户定义函数（user-defined function, UDF）、扩展现有的MySQL命令、给MySQL服务器添加自己的SQL命令。这些技巧可以让MySQL系统更好地满足用户环境的具体需要。



系统集成商遇到的最大挑战之一是如何克服被集成系统的局限性。这通常是因为某个子系统本身有局限性，或没有集成工作所需要的某个函数或命令。这经常意味着需要创建更多“中介”（glue）程序来转换或扩展现有的函数和命令才能解决问题。

MySQL AB公司早就看到了这种需求，并在MySQL服务器里增加了可以让系统集成商用来添加新函数和新命令的灵活选项。比如说，你可能需要增加一些函数来完成某种计算或数据转换，或者可能需要一个新命令为系统管理工作提供某种特定的数据。本章将介绍几种为MySQL服务器添加新函数的办法，还将演示如何给MySQL服务器添加自己的SQL命令。学习本章所需要的大部分背景知识已经在前几章讨论过了，如果还有什么不明白的地方，请参阅前面的几章。

8.1 添加用户定义函数

用户定义函数（user-defined function，UDF）是MySQL很早以前就支持的。简单地说，UDF就是用户为MySQL服务器添加的新函数（计算、数据转换等），它们将成为MySQL自带函数清单里的新成员。UDF的最大优点是它们可以在运行时动态地加载。不仅如此，你还可以创建自己的UDF库并在企业里使用它们或是免费发行它们（作为开源）。许多系统集成厂商都把UDF作为扩展MySQL服务器功能的首选办法。MySQL中的动态加载/卸载UDF机制是MySQL AB公司的另一个创新之举。

这个机制与插入式接口很相似，事实上，后者就是在前者的基础上发展而来的。UDF接口使用外部的可动态加载目标文件来加载和卸载UDF。这个机制使用CREATE FUNCTION命令来建立一条与可加载目标文件的连接（每次加载一个函数），使用DROP FUNCTION命令来卸载UDF。我们先来看看这两条命令的语法。

8.1.1 CREATE FUNCTION 命令的语法

CREATE FUNCTION命令用来向服务器注册函数，它将在给定数据库的func表里添加一个行。这条命令的语法如下所示。

```
CREATE FUNCTION function_name RETURNS [STRING | INTEGER | REAL] SONAME "mylib.so";
```

function_name代表正在创建的函数的名字。返回类型可以是STRING、INTEGER或REAL之一。SONAME是包含着这个函数的库文件的名称（*.so或*.dll）。这些库文件包含有关函数的源代码。它们通常是一些C函数，然后被编译成一个目标文件。CREATE FUNCTION命令将使得MySQL服务器在命令里的函数名（function_name）与给定的目标文件之间创建一个映射关系。当执行到那个函数的时候，服务器将

调用库文件里的函数代码去执行。

8.1.2 DROP FUNCTION 命令的语法

DROP FUNCTION命令用来向服务器注销一个函数，它将从给定数据库的func表里删除相关的行。这条命令的语法如下所示，function_name代表着你正在注销的函数的名字。

```
DROP FUNCTION function_name;
```

用户定义函数可以用在SQL语言允许使用表达式的任何地方。比如说，你在存储过程和SELECT语句里都可以使用UDF。如果你正在寻找一种不需要修改MySQL源代码就能扩展MySQL服务器功能的办法，UDF将是最佳选择。事实上，你可以创建任意多个UDF，还可以把它们集中起来构成UDF库。UDF库是从源代码文件编译出来的二进制可执行文件（Linux平台上的.so文件，Windows平台上的.dll文件）。接下来，一起去看看如何创建一个UDF库和如何在自己的MySQL服务器里使用它。

8.1.3 创建用户定义库

用户定义函数分为两大类。

- ❑ 单次调用型，这种UDF与普通意义上的函数没什么区别，它们接受一些输入参数并返回一个结果。
- ❑ 多次调用型，这种函数本身与普通意义上的函数没什么区别，但它们往往会在一次查询里被调用许多次。比如说，可以创建一个UDF把一种数据类型转换为另一种，比如把一个日期字段从一种格式转换为另一种；你还可以创建一个函数来对一组记录进行某种高级的运算，比如求它们的平方和。UDF只能返回整数、字符串或实数值。

单次调用型UDF是最常见的。它们用来根据一个或多个参数完成某个操作。在某些场合，它们可以不需要输入参数。比如说，你可以创建一个UDF来返回某个全局状态变量（如SERVER_STATUS）的值。这种类型的UDF通常用在SELECT语句的字段清单里，或是用在存储过程里作为辅助函数。

多次调用型UDF用在GROUP BY子句里。如果你在GROUP BY子句里使用了一个UDF，服务器会为表里的每个行调用它一次，最后在整个分组的末尾还会再调用它一次。

创建UDF库的过程是创建新项目来提供UDF的加载/卸载方法（xxx_init()和xxx_deinit()，“xxx”是函数的名字）和函数本身的过程。xxx_init()和xxx_deinit()函数只需为每个语句调用一次。如果你打算创建一个多次调用型函数，还需要实现分组函数xxx_clear()和xxx_add()。如果你在GROUP BY子句里使用了一个函数，MySQL服务器将在分组处理的开始调用xxx_clear()函数来重置或归零有关的值，然后为分组里的每一个行分别调用一次xxx_add()函数，在分组处理的最后再调用一次这个函数本身。这一系列调用将完成以下工作：清除上一次的统计结果，对本次分组里的数据进行统计，最后调用这个函数本身返回统计结果。

一旦实现了这些函数，就可以编译该文件并把它复制到你的服务器安装目录的bin子目录下。可以通过CREATE FUNCTION命令来加载和使用这些函数。

注解 如果没有特别说明，MySQL源代码文件在Unix/Linux平台上都以.cc为扩展名，在Windows平台上则都以.cpp为扩展名。

MySQL AB公司提供了一个样板UDF项目，你打算创建的任何类型的函数都可以在这个样板项目里找到参考例子。这为你添加自己的函数提供了一个最佳的出发点。这个样板UDF项目提供的样板函数包括。

- 一个 metaphon 函数，用来对字符串进行类似 soundex 的处理。
- 一个示例函数，对参数值里的各个字符的编码进行累加，再除以所有参数的长度之和，把这个除法运算的商返回为一个双精度浮点数。
- 一个示例函数，对各个参数的长度进行累加，把它们的和返回为一个整数值。
- 一个 sequence 函数，对一组整数参数（数量）和双精度浮点数参数（成本）求取它们的平均值。
- 一个 aggregate 示例函数，返回整型参数（数量）和双精度型参数（成本）的平均成本。

根据你的需要，你可能会发现这些例子很有用。

让我们从复制这个样板UDF项目开始。先在MySQL源代码树的根目录里创建一个名为expert_udf的新文件夹，再从MySQL源代码树根目录下的/examples/udf_example子目录里找到udf_example.cc文件并把它复制到expert_udf子目录里。把那个文件创新命名为expert_udf.cc。

注解 有些MySQL源代码发行版本把这些文件放在/sql子目录里。如果是这样，你可能不需要修改制作文件。

如果你使用的是Linux平台，还需要把制作文件从/examples/udf_example子目录复制到新建的/expert_udf子目录里。你将需要打开这些文件并对路径名和文件名进行必要的替换（比如说，把udf_example替换为expert_udf）。虽然udf_example文件收录在了MySQL源代码的某些发行版本里，但用户通常应该把这作为一个单独的项目来编译。编译expert_udf文件并把它复制到你的MySQL服务器安装里。可以用以下命令来编译这个文件。

```
gcc -shared -o expert_udf.so expert_udf.cc -I/usr/local/mysql/include/mysql
```

注解 为了在Linux平台上使用UDF，必须使用动态链接库来编译你的MySQL服务器：在编译前的configure命令里给出-with-mysql-ldflags=-rdynamic开关。

```

VERSION      1.0
EXPORTS
    metaphon_init
    metaphon_deinit
    metaphon
    myfunc_double_init
    myfunc_double
    myfunc_int
    myfunc_int_init
    sequence_init
    sequence_deinit
    sequence
    avgcost_init
    avgcost_deinit
    avgcost_reset
    avgcost_add
    avgcost_clear
    avgcost

```

注意 Windows用户必须把与网络有关的UDF从udf_example库里删掉，因为Windows不能直接支持那些UDF。如果在编译时出现“未找到某某头文件”或“未找到某某外部函数”错误，请把相应的函数或语句改成注释。

如果你在编译时遇到错误，就要回去改正它们。最常见的原因是你替换文件名时有遗漏，或是在设置头文件路径时出现了拼写错误。

库编译好以后，我们来测试一下它的加载和卸载操作。这可以检查出对它的编译是否正确，以及它是否已被安装到了正确的地点。打开一个MySQL客户端窗口，发出CREATE FUNCTION和DROP FUNCTION命令加载/卸载这个UDF库里的所有函数。代码清单8-2给出了加载和卸载这个库文件里的前5个函数的命令。这份清单是供Windows平台使用的，Linux用户要把expert_udf.dll替换为expert_udf.so。

代码清单8-2 CREATE FUNCTION和DROP FUNCTION命令示例

```

CREATE FUNCTION metaphon RETURNS STRING SONAME "expert_udf.dll";
CREATE FUNCTION myfunc_double RETURNS REAL SONAME "expert_udf.dll";
CREATE FUNCTION myfunc_int RETURNS INTEGER SONAME "expert_udf.dll";
CREATE FUNCTION sequence RETURNS INTEGER SONAME "expert_udf.dll";
CREATE AGGREGATE FUNCTION avgcost RETURNS REAL SONAME "expert_udf.dll";

DROP FUNCTION metaphon;
DROP FUNCTION myfunc_double;
DROP FUNCTION myfunc_int;
DROP FUNCTION sequence;
DROP FUNCTION avgcost;

```

代码清单8-3和代码清单8-4给出了代码清单8-2里的CREATE FUNCTION和DROP FUNCTION命令的正确结果。

代码清单8-3 加载函数

```
mysql> CREATE FUNCTION metaphon RETURNS STRING SONAME "expert_udf.dll";
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE FUNCTION myfunc_double RETURNS REAL SONAME "expert_udf.dll";
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE FUNCTION myfunc_int RETURNS INTEGER SONAME "expert_udf.dll";
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE FUNCTION sequence RETURNS INTEGER SONAME "expert_udf.dll";
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE AGGREGATE FUNCTION avgcost RETURNS REAL SONAME "expert_udf.dll";
Query OK, 0 rows affected (0.00 sec)
```

代码清单8-4 卸载函数

```
mysql> DROP FUNCTION metaphon;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP FUNCTION myfunc_double;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP FUNCTION myfunc_int;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP FUNCTION sequence;
Query OK, 0 rows affected (0.00 sec)

mysql> DROP FUNCTION avgcost;
Query OK, 0 rows affected (0.00 sec)
```

接下来，我们来看看这些命令能否正常工作。回到MySQL客户端窗口，再次运行代码清单8-2里的CREATE FUNCTION命令加载那些UDF。代码清单8-5给出了这个库里的前5个UDF函数的执行情况。你可以随意试用这些命令，看到的结果应该与这份清单差不多。

代码清单8-5 UDF命令的执行示例

```
mysql> SELECT metaphon("This is a test.");
```

```
+-----+
| metaphon("This is a test.") |
+-----+
| OSSTS                        |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT myfunc_double(5.5, 6.1);
```

```
+-----+
| myfunc_double(5.5, 6.1) |
+-----+
| 50.17                   |
+-----+
1 row in set (0.01 sec)
```

```
mysql> SELECT myfunc_int(5, 6, 8);
```

```
+-----+
| myfunc_int(5, 6, 8) |
+-----+
| 19                   |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT sequence(8);
```

```
+-----+
| sequence(8) |
+-----+
| 9           |
+-----+
1 row in set (0.00 sec)
```

```
mysql> CREATE TABLE testavg (order_num int key auto_increment, cost double,
mysql> qty int);
```

```
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> INSERT INTO testavg (cost, qty) VALUES (25.5, 17);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO testavg (cost, qty) VALUES (0.23, 5);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO testavg (cost, qty) VALUES (47.50, 81);
```

```
Query OK, 1 row affected (0.00 sec)
```

```
mysql> SELECT avgcost(qty, cost) FROM testavg;
```



```

+-----+
| avgcost(qty, cost) |
+-----+
| 41.5743             |
+-----+
1 row in set (0.03 sec)

```

最后几条命令演示了avgcost()统计函数的一个基本用法。统计函数通常都用在GROUP BY子句里，但这类函数有的也可以用在SELECT语句的列清单里，对表里的值进行分析。

8.1.4 添加新的用户定义函数

我们来给这个库添加一个新的UDF。不妨假设你正在开发一个系统集成项目，它需要你日期表示为Julian格式。Julian日期只计算年和日（从上一年的12月31日算起经过的天数），没有月的概念，把年和日两部分拼凑成一个DDDYYYY形式的数值就得到了Julian格式的日期。现在，需要编写一个函数把一个给定的日期（年、月、日）转换为Julian日期。下面是这个函数的定义。

```
longlong julian(int month, int day, int year);
```

为了简化这个函数的编写工作，我使用了3个整数作为输入参数。这个函数还可以实现成其他形式（比如接受日期值或字符串值作为输入参数等）。接下来，把JULIAN函数添加到刚才创建的UDF库里去。

这正体现了自行创建一个UDF库的好处：当需要添加一个新函数时，只要把它添加到现有的库文件里就行了，用不着再从头开始创建一个新项目。

添加一个新UDF的基本过程是这样的：先在UDF库源代码的extern节里添加对新函数的声明，然后实现新函数，最后重新编译那个库并把它部署到MySQL服务器的bin子目录里去。接下来，我们将按照这一过程来添加JULIAN函数。

打开expert_udf.cc文件并添加必要的函数声明。需要为julian_init()、julian_deinit()和julian()添加函数声明。julian_init()函数有3个输入参数。

□ UDF_INIT，一个用来在UDF方法之间传递信息的结构。

代码清单8-6 JULIAN函数的extern声明 (expert_udf.cc文件)

```
extern "C" {
my_bool julian_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
longlong julian(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error);
void julian_deinit(UDF_INIT *initid);
...
}
```

接下来是添加这些函数的实现代码。这里有个偷懒的办法：根据julian_xxx()函数的返回类型，从example_udf库文件里复制一个函数过来，然后根据需要对它进行修改。julian_init函数负责对变量进行初始化和检查语法错误。因为JULIAN函数要求它的3个输入参数都是整数，所以还需要加上一些出错处理代码对此进行检查。代码清单8-7给出了julian_init()函数的源代码。

代码清单8-7 julian_init()函数的源代码 (expert_udf.cc文件)

```
my_bool julian_init(UDF_INIT *initid, UDF_ARGS *args, char *message)
{
    if (args->arg_count != 3) /* if there are not three arguments */
    {
        strcpy(message, "Wrong number of arguments: JULIAN() requires 3 arguments.");
        return 1;
    }
    if ((args->arg_type[0] != INT_RESULT) ||
        (args->arg_type[1] != INT_RESULT) ||
        (args->arg_type[2] != INT_RESULT))
    {
        strcpy(message, "Wrong type of arguments: JULIAN() requires 3 integers.");
        return 1;
    }
    return 0;
}
```

在代码清单8-7里，先检查输入参数的个数是不是3个，然后检查它们的类型：这是为了确保它们都是整数。有经验的程序员会注意到，这段代码没有对输入参数的取值范围进行检查，而这有可能导致奇怪或非法的返回值。我把这个检查留给读者作为练习——如果你想在自己的UDF库里添加JULIAN函数的话。作为一个基本原则，只要某个输入参数的取值范围是已知的，就应该检查实际传递来的参数值是不是在那个范围内。

具体到这个例子，我们没有必要实现julian_deinit()函数，因为没有需要清理的内存或变量。但为了保持这个过程的完整性，你应该用一个空函数把它实现出来。作为一个基本原则，有些函数即使使用不着，也应该把它实现出来以保证整个过程的完整性。代码清单8-8给出了这个函数的源代码。因为没有用到任何新的变量或结构，所以它只是一个空函数。如果在julian_init()函数里创建过新的变量或结构，则需要在这个函数里释放它们。

代码清单8-8 julian_deinit()函数的源代码 (expert_udf.cc文件)

```
void julian_deinit(UDF_INIT *initid)
{
}
```

JULIAN函数的具体工作发生在julian()函数里。代码清单8-9给出了julian()函数的源代码。

注解 有一些复杂的Julian日历方法会从一个很早的日期（通常是18或19世纪的某一天）开始计算经过的天数。因为我只是想通过这个例子来演示如何添加一个新的UDF函数，所以这里给出的JULIAN函数只计算从上一年的12月31日开始经过的天数。

代码清单8-9 julian()函数的源代码 (expert_udf.cc文件)

```
longlong julian(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    longlong jdate = 0;
    static int DAYS_IN_MONTH[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    int month = 0;
    int day = 0;
    int year = 0;
    int i;

    /* copy memory from the arguments */
    memcpy(&month, args->args[0], args->lengths[0]);
    memcpy(&day, args->args[1], args->lengths[1]);
    memcpy(&year, args->args[2], args->lengths[2]);

    /* add the days in the month for each prior month */
    for (i = 0; i < month - 1; i++)
        jdate += DAYS_IN_MONTH[i];

    /* add the day of this month */
    jdate += day;

    /* find the year */
    if (((year % 100) != 0) && ((year % 4) == 0))
        jdate++; /*leap year!*/

    /* shift day of year to left */
    jdate *= 10000;

    /* add the year */
    jdate += year;
    return jdate;
}
```

请注意紧跟在变量声明部分后面的那几行代码，这是一个如何把参数值从args数组传递到本地变量的好例子，它们把前3个输入参数转换成了整数值，接下来的代码负责计算Julian日期并把它返回给调用者。

如果你使用的是Windows平台，还需要在expert_udf.def文件里添加3个函数名：julian_init、julian_deinit和julian。如代码清单8-10所示。

代码清单8-10 expert_udf.def文件的源代码

```

LIBRARY    MYUDF
DESCRIPTION 'MySQL Sample for UDF'
VERSION    1.0
EXPORTS
    metaphon_init
    metaphon_deinit
    metaphon
    myfunc_double_init
    myfunc_double
    myfunc_int
    myfunc_int_init
    sequence_init
    sequence_deinit
    sequence
    avgcost_init
    avgcost_deinit
    avgcost_reset
    avgcost_add
    avgcost_clear
    avgcost
    julian_init
    julian_deinit
    julian

```

接下来，编译这个UDF库，然后把它复制到MySQL服务器的bin子目录里：使用Linux平台的程序员需要从/expert_udf/debug子目录复制expert_udf.so文件，使用Windows平台的程序员需要从/expert_udf/debug子目录复制expert_udf.dll文件。

你应该在复制文件之前先关闭MySQL服务器，等复制操作完成后再重新启动它。这是因为（根据你把新函数放在什么地方）这次编译出来的目标文件可能与以前编译的不一样。作为一条基本原则，只要你对可执行代码做出了修改，就应该按照“关闭服务器—安装新代码—重新启动服务器”的步骤来做。

安装好新的UDF库之后，就可以输入CREATE FUNCTION命令去试试JULIAN函数的效果了。代码清单8-11给出了一个在Windows平台上加载和使用JULIAN函数的例子。

代码清单8-11 加载和使用julian()函数

```
mysql> CREATE FUNCTION julian RETURNS INTEGER SONAME "expert_udf.dll";
```

```
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> SELECT JULIAN(7, 4, 2006);
```

```

+-----+
| JULIAN(7, 4, 2006) |
+-----+
| 1852006             |
+-----+
1 row in set (0.00 sec)

```


你可以使用expert_udf库作为UDF库的起点，也可以按照本节给出的步骤复制它并创建自己的UDF库。UDF库可以帮助你扩展MySQL服务器的能力，以满足几乎所有的计算需要。除了需要有一个可动态加载的Linux版本外，UDF不需要我们对系统配置做什么调整就可以工作得很好。

8.2 添加本机函数

这里所说的“本机函数”是指那些被编译为MySQL服务器一部分的函数。它们不需要从某个库文件加载就可以在SQL命令里直接使用。MySQL的本机函数可以列成一个非常长的名单，从ABS()到UCASE()应有尽有。如果你想了解更多关于MySQL本机函数的信息，请查阅网上的《MySQL参考手册》。

如果你想使用的函数不存在（不是一个内建在MySQL服务器里的本机函数），可以通过修改MySQL源代码的办法去添加自己的本机函数。比如说，既然我们已经有了一个JULIAN函数，再有一个能把Julian日期转换回Gregorian日期的函数岂不是更好？本节将演示如何添加一个新的本机函数。添加一个新的本机函数需要修改几个MySQL源代码文件，我把需要修改的文件汇总在了表8-1里。

表8-1 添加一个新的本机函数需要修改的MySQL源代码文件

文 件	修改说明
lex.h	为新函数在词法解析器的符号表里添加一些符号
item_create.h	添加create_func_xxx函数的声明
item_create.cc	添加create_func_xxx函数的源代码
item_str_func.h	添加新函数的类定义
item_str_func.cc	添加新函数的类源代码
lex_hash.h	为词法解析器和语法解析器里的符号重新生成一份词法散列表

注解 表8-1里的文件都在MySQL源代码树根目录下的/sql子目录里。

对lex.h文件的修改包括为新函数添加一些符号，以及定义create_func_xxx函数的名字。MySQL服务器在处理SQL命令时将调用这个函数来创建一个新函数（类）的实例。打开lex.h文件，把如下所示的符号定义添加到symbols[]数组里。请注意，这个数组里的元素是按字母表顺序排列的，所以你也必须按照这个顺序来插入新的符号定义。

```
{ "GREGORIAN", F_SYM(FUNC_ARG1),0,CREATE_FUNC(create_func_gregorian)},
```

这个符号定义项的值依次是：新函数的名字、一个用来与解析器里的记号建立关联关系的F_SYM()函数调用、这个名字的长度（在sql_lex.cc文件里设置）、与新符号对应的创建函数的引用指针。这些数据项用来创建被识别出来的符号与代码里的位置之间的对应关系。

这个数组是如何告诉解析器在识别出某个符号之后应该做些什么的呢？这里使用了一种被称为词法散列表（lexical hash）的机制。如果你编译这些代码并运行它们，你将发现新符号没有被识别出来。这是因为你必须生成一份新的词法散列表，才能让新增加的符号被识别出来。MySQL中的词法散列表是Knuth提出的一种高级散列查询算法^①的一种实现。它需要用一个实现了那种算法的命令行工具来生成。这个工具的名字是gen_lex_hash，它的源代码文件是gen_lex_hash.cc。你将需要用这个工具

① Knuth D.E., *The Art of Computer Programming* .2nd ed, Addison-Wesley出版公司1997年出版。

生成的文件去替代现有的词法散列表头文件 (lex_hash.h)。在修改MySQL源代码的时候,一定要记住这样一条规则:只要修改了lex.h文件里的代码和/或添加了新的符号,就必须用gen_lex_hash工具重新生成lex_hash.h文件。本节后面的内容将向大家介绍在Windows和Linux平台上生成lex_hash.h文件的步骤。

既然已经告诉词法解析器有了一个新的创建函数,就需要提供一个函数声明。打开item_create.h文件,把如下所示的函数声明添加进去。“Item* a”参数是一个指向这个函数的输入参数的指针。

```
Item *create_func_gregorian(Item* a);
```

接下来是添加这个创建函数的源代码,这些代码用来对马上就要创建的Item_func_gregorian类进行实例化。打开item_create.cc文件,把代码清单8-12里的代码添加进去。

代码清单8-12 修改item_create.cc文件

```
Item *create_func_gregorian(Item* a)
{
    return new Item_func_gregorian(a);
}
```

把这个创建函数实现出来以后,还需要创建一个新的类来实现你的新本机函数的代码。因为这里的修改将影响到整个MySQL系统,MySQL AB公司提供了许多Item_xxx_func基类(和派生类)来帮助那些深感困惑的程序员。如果新本机函数返回一个字符串,应该从Item_str_func基类派生你的类;如果新本机函数返回一个整数,应该从Item_int_func基类派生你的类。类似的,返回其他数据类型的函数也有相应的基类可供选用。这是本机函数机制与可动态加载的UDF接口的分水岭,也是人们选择创建一个本机函数而不是选择创建一个动态可加载函数的主要原因。如果你想了解更多关于Item_xxx_func类的信息,请到MySQL源代码树根目录下的/sql子目录里查看item.h文件。

因为gregorian()函数将返回一个字符串,需要从Item_str_func基类派生你的类,在item_str_func.h文件里定义这个类,在item_str_func.cc文件里实现这个类。打开item_str_func.h文件,把代码清单8-13里的类定义添加到这个头文件里。请注意,这个类只有4个必须声明的函数。从Item_str_func基类派生出来的子类至少需要包括4个方法:一个包含着这个函数的代码的函数(Item_func_gregorian)、一个值函数(val_str)、一个用来返回名字的函数(func_name),以及一个用来为将被返回的字符串值设置最大长度的函数(fix_length_and_dec)。你还可以根据需要添加其他函数,但这4个函数是返回字符串的函数必须有的。其他的Item_xxx_func基类(和派生类)可能需要包括更多的函数如val_int()、val_double()等。如果你需要从某个基类派生一个子类,请查看它的类定义去了解必须包括哪些方法;这些方法被称为“虚函数”(virtual function)。

代码清单8-13 修改item_str_func.h文件

```
class Item_func_gregorian :public Item_str_func
{
public:
    Item_func_gregorian(Item *a) :Item_str_func(a) {}
    String *val_str(String *str);
    const char *func_name() const { return "gregorian"; }
```

```
void fix_length_and_dec();
};
```

接下来是添加类实现。打开item_str_func.cc文件，把代码清单8-14里的Gregorian类的函数实现添加进去。需要实现的主函数val_str()，负责具体完成从Julian格式到Gregorian格式的转换工作。还需要实现fix_length_and_dec()函数，为将被返回的字符串设置一个最大长度。

代码清单8-14 修改item_str_func.cc文件

```
String *Item_func_gregorian::val_str(String *str)
{
    static int DAYS_IN_MONTH[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    longlong jdate = args[0]->val_int();
    int year = 0;
    int month = 0;
    int day = 0;
    int i;
    char cstr[30];
    cstr[0] = 0;
    str->length(0);

    /* get date from value (right 4 digits */
    year = jdate - ((jdate / 10000) * 10000);

    /* get value for day of year and find current month*/
    day = (jdate - year) / 10000;
    for (i = 0; i < 12; i++)
        if (DAYS_IN_MONTH[i] < day)
            day = day - DAYS_IN_MONTH[i]; /* remainder is day of current month */
    else
    {
        month = i + 1;
        break;
    }

    /* format date string */
    sprintf(cstr, "%d", month);
    str->append(cstr);
    str->append("/");
    sprintf(cstr, "%d", day);
    str->append(cstr);
    str->append("/");
    sprintf(cstr, "%d", year);
    str->append(cstr);
    if (null_value)
        return 0;
    return str;
}

void Item_func_gregorian::fix_length_and_dec()
```

```
{  
    max_length=30;  
}
```

需要添加的东西都添加完了。下一个任务是生成词法散列表。请阅读与你的操作系统有关的章节。这一步非常关键，不管重复多少次，都必须保证编译出来的gen_lex_hash命令行工具没有任何错误。

8.2.1 在 Windows 平台上生成词法散列表

首先，打开主解决方案并把/sql子目录里的gen_lex_hash项目添加到其中。然后，把这些库添加到项目依赖关系dbug、libmysql、mysys、strings、taocrypt、yassl和zlib中。接下来编译这个项目。编译器将把没有处理过的库文件包括进去。编译完成后，打开一个命令窗口，进入MySQL源代码根目录下的/sql子目录，然后运行gen_lex_hash.exe程序生成词法散列表，如下所示：

```
gen_lex_hash > lex_hash.h
```

这将生成一个新的lex_hash.h文件，用它编译出来的MySQL服务器才能识别出你刚才添加到lex.h文件里的符号。

8.2.2 在 Linux 平台上生成词法散列表

Linux用户应该感到高兴，因为MySQL服务器的编译脚本可以替你们完成编译gen_lex_hash工具的工作。有几个MySQL源代码发行版本（5.1或更高的版本）把这个工具放在/sql子目录里，随mysqld服务器一同编译；其他发行版本（包括一些适用于Windows平台的发行版本）把这个工具放在一个名为gen_lex_hash子目录里并有它们自己的制作文件。运行gen_lex_hash工具就可以生成词法散列表，如下所示。

```
gen_lex_hash > lex_hash.h
```

这将生成一个新的lex_hash.h文件，用它编译出来的MySQL服务器才能识别刚才添加到lex.h文件里的符号。不过，其实用不着进行这个步骤，因为适用于Linux平台的MySQL源代码发行版本里的制作文件会替你完成这个步骤。如果你想运行这个命令，唯一的理由大概是你想看看编译过程有没有错误吧。

8.2.3 编译和测试新的本机函数

重新编译服务器并重新启动它。如果在编译时遇到错误，请检查刚才添加的语句有没有错误。改正错误并编译出一个新的可执行文件后，关闭服务器，把新的可执行文件复制到你的MySQL安装目录里，然后重新启动服务器。现在，可以执行新的Gregorian本机函数看看它的效果了，如代码清单8-15和代码清单8-16所示。如果你想测试一下Gregorian函数的正确性，可以先运行julian()函数，再用它的输出作为输入去运行gregorian()函数。

代码清单8-15 运行julian()函数

```
mysql> select julian(7,4,2006);
```



```

+-----+
| julian(7,4,2006) |
+-----+
| 1852006          |
+-----+
1 row in set (0.00 sec)

```

代码清单8-16 运行gregorian()函数

```
mysql> select gregorian(1852006);
```

```

+-----+
| gregorian(1852006) |
+-----+
| 7/4/2006           |
+-----+
1 row in set (2.44 sec)

```

好了，添加一个新本机函数的工作就此圆满结束。在掌握了创建本机函数的技巧之后，读者在制定系统集成方案的时候又多了一种选择——定制MySQL服务器的源代码。

8.3 添加 SQL 命令

如果现有的SQL命令不能满足你的需要，用户定义函数也解决不了你的问题，不妨考虑一下给MySQL服务器添加新SQL命令的办法。本节将向大家演示如何为MySQL服务器添加自己的SQL命令。

有不少人认为，在扩展MySQL服务器功能的各种办法当中，通过修改MySQL的源代码给它添加新的SQL命令是最困难的。但正如你们将看到的那样，这个过程与其说是复杂，不如说是繁琐。要想添加新的SQL命令，必须修改解析器（sql\sql_yacc.yy）并把新命令添加到SQL命令处理代码（sql\sql_parse.cc）里去。

当客户端发出一个查询时，MySQL服务器将创建一个新的线程，并把SQL命令传递到解析器进行语法检查。MySQL解析器是用一个非常大的用Bison工具编译得到的Lex-YACC脚本实现出来。解析器将构造一个用来在内存里代表查询语句（SQL）的查询结构，这个结构可以用来执行查询。要想给解析器添加一个新的命令，你手里必须有GNU Bison工具才行。可以从GNU网站^①下载Bison工具并安装。

Lex、YACC和Bison

Lex的含义是词法解析器生成器（lexical analyzer generator），它是一个用来把语句/表达式分解成一系列最基本的记号和常数的解析器，也可以完成一些语法检查工作。YACC的含义是一个编译器的编译器（yet another compiler compiler），它可以根据你用YACC代码写出来的语法定义为你的

① Linux用户可以从GNU网站www.gnu.org/software/bison下载。Windows用户可以从<http://gnuwin32.sourceforge.net/packages/bison.htm>下载一个Win32版本。

“新”语言生成一个编译器。这两个工具，再加上Bison（一个能够从Lex/YACC代码生成C源代码的YACC编译器）——可以帮助人们简便快速地开发出一个能够分析和处理语言命令的子系统。MySQL的解析器就是一个典型的例子。

不妨假设你打算给MySQL服务器增加一个新命令，它可以把MySQL服务器里的各个数据库的磁盘空间占用量显示给你查看。你知道有些外部工具可以检索出这个信息，但需要的是一个能够完成同样工作的SQL命令。不妨再假设你想把这个功能添加为一个SHOW命令。具体地说，你想添加一条SHOW DISK_USAGE命令，它将把MySQL服务器里的各个数据库和各数据库里所有文件（表）的总长度（以KB为单位）列成一份清单显示。

添加一条新的SQL命令需要在词法解析器里添加一些符号，还需要把SHOW DISK_USAGE命令的语法添加到YACC解析器（sql_yacc.yy）里。新的解析器需要用Bison工具编译成一个C程序，还需要用gen_lex_hash工具生成一份新的词法散列表。你还需要在sql_parse.cc文件里为新命令添加一个case语句，这样才能让解析器把控制转到新命令。

我们就从给词法解析器里添加一些符号开始吧。打开lex.h文件并找到static SYMBOL symbols[]数组。这个符号可以随意选择，但最好使用一个有意义的单词（就像那些好的变量名那样）。这里需要注意的是，不要选择一个已在使用的符号。具体到这个例子，我选择的符号是DISK_USAGE。它的作用相当于一个标签，解析器将把它识别为一个记号。在这个数组里插入一条语句以便让词法解析器生成这个符号，我把那个记号命名为DISK_USAGE_SYM。这个数组里的元素基本上是按照字母表顺序排列的，所以必须把它放到一个适当的位置。代码清单8-17给出了增加这些符号后的symbols[]数组的片段。

代码清单8-17 为SHOW DISK_USAGE命令更新lex.h文件

```
static SYMBOL symbols[] = {
    { "&&",    SYM(AND_AND_SYM)},
    ...
    { "DISK",    SYM(DISK_SYM)},
    /* BEGIN CAB MODIFICATION */
    /* Reason for Modification: */
    /* This section identifies the tokens for the SHOW DISK_USAGE command*/
    { "DISK_USAGE",    SYM(DISK_USAGE_SYM)},
    /* END CAB MODIFICATION */
    { "DISTINCT",    SYM(DISTINCT)},
    ...
}
```

你需要做的下一件事是增加一个助记符来标识这个命令。这个助记符将在解析器里用来创建和标识一个内部查询结构，并通过sql_parse.cc文件里的超大型switch语句的一个case分支对执行流程进行控制。打开sql_lex.h文件并把新命令添加到enum_sql_command枚举集合里。代码清单8-18给出了为新命令添加助记符后的sql_lex.h文件的代码片段。

代码清单8-18 为SHOW DISK_USAGE命令修改sql_lex.h文件

```
enum enum_sql_command {
    SQLCOM_SELECT, SQLCOM_CREATE_TABLE, SQLCOM_CREATE_INDEX, SQLCOM_ALTER_TABLE,
```

```

SQLCOM_UPDATE, SQLCOM_INSERT, SQLCOM_INSERT_SELECT,
SQLCOM_DELETE, SQLCOM_TRUNCATE, SQLCOM_DROP_TABLE, SQLCOM_DROP_INDEX,
...
SQLCOM_SHOW_COLUMN_TYPES, SQLCOM_SHOW_STORAGE_ENGINES, SQLCOM_SHOW_PRIVILEGES,
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section captures the enumerations for the SHOW DISK_USAGE command tokens */
SQLCOM_SHOW_DISK_USAGE,
/* END CAB MODIFICATION */
SQLCOM_HELP, SQLCOM_CREATE_USER, SQLCOM_DROP_USER, SQLCOM_RENAME_USER,
...

```

添加了新符号和新命令助记符以后，现在需要在sql_yacc.yy文件里添加一些代码来定义你在lex.h文件里使用的新记号，还需要为新的SQL命令SHOW DISK_USAGE添加源代码。打开sql_yacc.yy文件并把新记号添加到记号表（在文件的开头部分）里。这个记号表里的元素基本上是按照字母表顺序排列的，所以必须把它放到一个适当的位置。代码清单8-19给出了对sql_yacc.yy文件的修改情况。

代码清单8-19 把新记号添加到sql_yacc.yy文件里

```

...
%token DISK_SYM
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section defines the tokens for the SHOW DISK_USAGE command */
%token DISK_USAGE_SYM
/* END CAB MODIFICATION */
%token DISTINCT
...

```

注解 适用于Windows平台的MySQL源代码发行版本没有收录sql_yacc.yy文件。Windows程序员需要下载一份适用于Linux平台的MySQL源代码发行版本，然后从中提取出这个文件。请注意，你下载的MySQL源代码（Linux平台）的版本号必须与现有的MySQL源代码（Windows平台）保持一致。

还需要把新命令的语法添加到解析器的YACC代码里（也在sql_yacc.yy文件里）。找到show:标签，把代码清单8-20里的命令添加进去。

代码清单8-20 SHOW DISK_USAGE命令的解析器语法源代码

```

show:
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section captures (parses) the SHOW ALV statement */
SHOW DISK_USAGE_SYM
{
    LEX *lex=Lex;
    lex->sql_command= SQLCOM_SHOW_DISK_USAGE;
}
/* END CAB MODIFICATION */
| SHOW

```


注意 千万不要忘记在原来的SHOW语句前面加上一个|字符。

也许你正想知道这些代码是干什么用的。在解答这个疑问之前，我想强调一下：这个地方非常重要，有很多程序员就是因为这里出了错误而失败的。

Show: 标签所标识的那段代码是解析器在识别出SHOW记号后将要执行的代码。YACC代码几乎总是被编写成这个样子^①。SHOW DISK_USAGE_SYM语句给出了包含SHOW和DISK_USAGE这两个记号唯一正确的语法（必须按顺序出现）。如果浏览一下前后的代码，你将发现其他类似的语法安排。跟在语法语句后面的代码块先获得了一个指向lex结构的指针，然后把command属性设置为新命令记号SQLCOM_SHOW_DISK_USAGE。请注意这段代码是如何把SHOW和DISK_USAGE记号与SQLCOM_SHOW_DISK_USAGE命令关联起来的，这使得sql_parse.cc文件里的超大型switch语句可以正确地把执行流导向SHOW_DISK_USAGE命令的代码实现。

还请注意，我把这段代码放在了show: 定义的开头并在原来的SHOW语法语句的前面加上了一个垂线字符|。垂线字符是YACC语言里的“或”运算符。这意味着只有与show: 标记引导的语法语句定义之一相匹配的SHOW命令才是合法的（其他SQL命令也是这种情况）。你可以趁这个机会看看这个文件里的其他代码，这可以让你对那些代码的工作情况更加熟悉。不管学习什么，不放过每一个细节都是非常重要的。如果你打算实现更复杂的命令，可以在这个文件里找一些类似的命令作为例子，看它们是如何处理各有关记号和变量的。

接下来，需要把一些源代码添加到sql_parse.cc文件中的超大型switch语句里去。打开这个文件并在那个switch语句里增加一个新的case分支，如代码清单8-21所示。

代码清单8-21 为新命令增加一个case分支

```
...
case SQLCOM_SHOW_AUTHORS:
    res= mysqld_show_authors(thd);
    break;
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SHOW DISK_USAGE command. */
case SQLCOM_SHOW_DISK_USAGE:
    res = show_disk_usage_command(thd);
    break;
/* END CAB MODIFICATION */
case SQLCOM_SHOW_PRIVILEGES:
    res= mysqld_show_privileges(thd);
    break;
...
```

请注意，我只增加了一个对新函数show_disk_usage_command()的调用。我们将把这个函数的源代码添加到sql_show.cc文件里去。这个函数的名字与lex.h文件里的记号、sql_yacc.yy文件里的符号、sql_parse.cc文件里的case分支标签是相互对应的。这不仅有助于我们追踪有关代码的分布和流向，

^① 如果你想了解更多关于YACC解析器和如何编写YACC代码的信息，请自行查阅<http://dinosaur.compile-rttools.net/>网站上的有关资料。

还有助于让那个已经很大的switch语句易于管理。这里是MySQL服务器控制SQL命令执行情况的中枢，所以你应该趁这个机会阅读一下前后的代码，会看到SELECT、CREATE等所有的命令。

现在，我们来添加负责具体执行新命令的代码。打开mysql_priv.h文件，把代码清单8-22给出的新命令的函数声明添加进去。我把新函数的声明插入与sql_parse.cc文件里的case分支顺序相对应的位置——这不是必要的，但它可以让代码更容易阅读和跟踪。

代码清单8-22 为新命令添加一个函数声明

```
...
bool mysqld_show_authors(THD *thd);
bool show_disk_usage_command(THD *thd);
bool mysqld_show_privileges(THD *thd);
...
```

最后的修改是添加show_disk_usage_command()函数的代码实现（代码清单8-23）。打开sql_show.cc文件，把新命令的函数实现添加进去。代码清单8-23里的代码是“假”的，因为我想在添加更多的代码之前先行确认新命令真的能够工作。这个技巧很重要，尤其是在需要编写一大段非常复杂的代码的时候。先搭建一个基本的代码框架有许多好处，它不仅可以让你有机会检查编程思路是否正确，还可以让你在插入大段代码之后遇到问题时缩小调试范围——那些错误至少与你的框架无关。在修改现有的SQL命令或添加新的SQL命令时，这个技巧可以让你少走许多弯路，其重要性不言而喻。

代码清单8-23 show_disk_usage_command()函数的“假”实现

```
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SHOW DISK_USAGE command. */
bool show_disk_usage_command(THD *thd)
{
    List<Item> field_list;
    Protocol *protocol= thd->protocol;
    DBUG_ENTER("show_disk_usage");

    /* send fields */
    field_list.push_back(new Item_empty_string("Database",50));
    field_list.push_back(new Item_empty_string("Size (Kb)",30));
    if (protocol->send_fields(&field_list,
                            Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DBUG_RETURN(TRUE);

    /* send test data */
    protocol->prepare_for_resend();
    protocol->store("test_row", system_charset_info);
    protocol->store("1024", system_charset_info);
    if (protocol->write())
        DBUG_RETURN(TRUE);

    send_eof(thd);
}
```

```

    DEBUG_RETURN(FALSE);
}
/* END CAB MODIFICATION */

```

关于这段源代码，还有几句话要说。首先，我在本书前面的章节里曾经提到过，MySQL提供了一些底层的网络函数，用来创建结果集并把它返回给客户端。请看“/* send fields */”注释后面的那几行代码，它们负责为结果集创建字段。具体到这个例子，我创建了两个字段（列），一个名为Database，另一个名为Size (Kb)；它们将成为你在MySQL客户端工具里执行该命令之后所看到的列标题。

再看那些protocol->XXX语句，这些Protocol类的方法负责把行发送给客户端。我先调用了prepare_for_resend()方法清理缓冲区，然后调用了两次store()方法（这个方法有许多复用形式），为每个字段填充了一个数据值（按照字段的先后顺序）。最后，调用了write()方法把缓冲区的内容输出到了网络。我在这几个调用的后面还安排了一个DEBUG_RETURN(TRUE)标记——如果这个函数返回了TRUE，就会知道是这几个调用在执行时出了问题。最后一条语句send_eof()函数将结束结果集并结束与客户端的通信。在为MySQL服务器添加自己的新SQL命令时，可以按照这个套路使用这些类、方法和函数来发送结果。

如果你现在就去编译MySQL服务器，将遇到“DISK_USAGE_SYM符号未定义”之类的错误；这很正常，因为还有一些事情没有做。不过，如果这次编译还有其他错误，就说明你刚才添加的那些东西有问题，请把它们改正过来再继续。

现在，回到词法散列表和解析器代码。如果你曾研究过MySQL源代码，应该注意到有sql_yacc.cc和sql_yacc.h两个文件。这些文件是用Bison工具从sql_yacc.yy文件生成的。我们现在要做的就是用Bison工具生成这些文件。打开一个命令窗口，进入MySQL源代码树根目录下的/sql子目录，然后执行以下命令。

```
bison -y -d sql_yacc.yy
```

这将生成两个新文件：y.tab.c和y.tab.h，这些文件将分别替代sql_yacc.cc和sql_yacc.h文件。在复制它们之前，应该先为原来的文件制作一个备份。然后把y.tab.c复制为sql_yacc.cc（Windows用户需要复制为sql_yacc.cpp），把y.tab.h复制为sql_yacc.h。

注解 使用Linux平台的程序员不需要进行这一步，因为适用于Linux平台的MySQL源代码发行版本里的制作文件会替你完成这一步。如果你想运行这个命令，唯一的理由可能是想看看编译过程有没有错误吧。

Windows程序员在编译sql_yacc.cpp文件时，可能会遇到“某某符号未定义”错误。如果你遇到的错误是yyerror或MYSQLparse未定义，请打开sql_yacc.cpp文件把代码清单8-24里的语句添加进去，然后重新运行Bison命令生成y.tab.c和y.tab.h文件。

代码清单8-24 缺少的#define语句

```

/* If NAME_PREFIX is specified substitute the variables and functions
   names. */
#define yyparse MYSQLparse
#define yylex   MYSQLlex
#define yyerror MYSQLerror

```

```
#define yylval MYSQLlval
#define yychar MYSQLchar
#define yydebug MYSQLdebug
#define yynerrs MYSQLnerrs
```

有了正确的sql_yacc.cc和sql_yacc.h文件，就可以用下面这条命令去生成一份新的词法散列表了。

```
gen_lex_hash > lex_hash.h
```

现在，编译MySQL服务器的准备工作已全部完成。因为修改了一些关键的头文件，这次编译所花费的时间可能会比平时长一些。如果遇到编译错误，请改正之后再继续。

在编译完MySQL服务器并得到一个新的可执行文件后，关闭正在使用的MySQL服务器，把新的可执行文件复制到MySQL的安装目录，然后重新启动MySQL服务器。现在，你就可以在一个MySQL客户端工具里试试新命令了。代码清单8-25给出了尝试SHOW DISK_USAGE命令的一个例子。

代码清单8-25 试用SHOW DISK_USAGE命令

```
mysql> SHOW DISK_USAGE;
```

```
+-----+-----+
| Database | Size (Kb) |
+-----+-----+
| test_row | 1024      |
+-----+-----+
1 row in set (0.00 sec)
```

如果一切正常，打开sql_show.cc文件把代码清单8-26里的SHOW DISK_USAGE命令的实际代码添加进去。

代码清单8-26 show_disk_usage_command()函数的源代码

```
/* This section adds the code to call the new SHOW DISK_USAGE command. */
bool show_disk_usage_command(THD *thd)
{
    List<Item> field_list;
    List<char> dbs;
    char *db_name;
    char *path;
    MY_DIR *dirp;
    FILEINFO *file;
    longlong fsizes = 0;
    longlong lsizes = 0;
    Protocol *protocol= thd->protocol;
    DEBUG_ENTER("show_disk_usage");

    /* send the fields "Database" and "Size" */
    field_list.push_back(new Item_empty_string("Database",50));
    field_list.push_back(new Item_int("Size (Kb)",(longlong) 1,21));
    if (protocol->send_fields(&field_list,
                            Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DEBUG_RETURN(TRUE);
}
```

```

/* get database directories */
mysql_find_files(thd, &db, 0, mysql_data_home, 0, 1);
List_iterator_fast<char> it_db(db);
path = (char *)my_malloc(PATH_MAX, MYF(MY_ZEROFILL));
dirp = my_dir(mysql_data_home, MYF(MY_WANT_STAT));
fsizes = 0;
lsizes = 0;
for (int i = 0; i < (int)dirp->number_off_files; i++)
{
    file = dirp->dir_entry + i;
    if (strncasecmp (file->name, "ibdata", 6) == 0)
        fsizes = fsizes + file->mystat->st_size;
    else if (strncasecmp (file->name, "ib", 2) == 0)
        lsizes = lsizes + file->mystat->st_size;
}

/* send InnoDB data to client */
protocol->prepare_for_resend();
protocol->store("InnoDB TableSpace", system_charset_info);
protocol->store((longlong)fsizes);
if (protocol->write())
    DEBUG_RETURN(TRUE);
protocol->prepare_for_resend();
protocol->store("InnoDB Logs", system_charset_info);
protocol->store((longlong)lsizes);
if (protocol->write())
    DEBUG_RETURN(TRUE);

/* now send database name and sizes of the databases */
while (db_name = it_db++)
{
    fsizes = 0;
    strcpy(path, mysql_data_home);
    strcat(path, "/");
    strcat(path, db_name);
    dirp = my_dir(path, MYF(MY_WANT_STAT));
    for (int i = 0; i < (int)dirp->number_off_files; i++)
    {
        file = dirp->dir_entry + i;
        fsizes = fsizes + file->mystat->st_size;
    }
    protocol->prepare_for_resend();
    protocol->store(db_name, system_charset_info);
    protocol->store((longlong)fsizes);
    if (protocol->write())
        DEBUG_RETURN(TRUE);
}
send_eof(thd);

/* free memory */

```



```

my_free((gptr)path, MYF(0));
DEBUG_RETURN(FALSE);
}
/* END CAB MODIFICATION */

```

注解 如果你使用的是Windows平台，请把my_malloc()调用里的PATH_MAX替换为MAX_PATH，再把strncasecmp调用全部替换为strnicmp调用。

重新编译和安装MySQL服务器，然后运行这条新命令，你应该看到如代码清单8-27所示的结果。

代码清单8-27 执行新的SHOW DISK_USAGE命令

```
mysql> SHOW DISK_USAGE;
```

```

+-----+-----+
| Database      | Size (Kb) |
+-----+-----+
| InnoDB TableSpace | 10485760 |
| InnoDB Logs      | 20971520 |
| cluster        | 9867      |
| mysql           | 617310    |
| test           | 9720      |
+-----+-----+
5 rows in set (0.65 sec)

```

你将看到MySQL服务器里的每一个数据库的磁盘空间占用情况。美中不足的是，这份清单只列出了各个数据库的磁盘空间占用量，没有对它们进行总计（像WITH ROLLUP子句那样）。我把这个修改留给读者作为练习，好趁此机会试试自己刚学到的东西。

虽然篇幅不长，但我希望本节内容能对你解决创建新的SQL命令时遇到的问题有所帮助。有了这些知识，在制定系统集成方案的时候就又多了一种选择——扩展/创建MySQL命令。

8.4 添加到信息模式

本章要讨论的最后一个问题是如何把信息添加到MySQL信息模式中。信息模式（information schema）是对一组驻留在内存里的逻辑表的统称，该逻辑表包含着关于MySQL服务器及其运行环境的状态信息和各种统计信息（也叫元数据）。信息模式始见于MySQL 5.0.2版，它现在已经是对MySQL服务器、它的运行环境和数据库进行管理和调试的重要工具之一^①。比如说，利用信息模式，我们只需发出如下所示的命令，就可以轻而易举地查看到某给定数据库里所有表的所有列的清单。

```

SELECT table_name, column_name, data_type FROM information_schema.columns
WHERE table_schema = 'test';

```

把元数据分门别类地组织成一些逻辑表的最大好处是我们可以用SELECT命令去查询它们。这种机制为数据库系统管理员提供了一种独特而又高效的收集关于MySQL服务器的信息的手段。

表8-2列出了一些常用的逻辑表和它们的用途。

① 如果你想了解更多信息模式的信息，请参阅网上的《MySQL参考手册》。

表8-2 常用的MySQL信息模式

名 称	说 明
schemata	提供关于数据库的信息
tables	提供关于所有数据库里的表的信息
columns	提供关于所有表里的列的信息
statistics	提供关于各表的索引的信息
user_privileges	提供关于数据库操作权限的信息，它封装着mysql.db权限表
table_privileges	提供关于表操作权限的信息，它封装着mysql.tables_priv权限表
column_privileges	提供关于列操作权限的信息，它封装着mysql.columns_priv权限表
collations	提供关于字符集和排序方式的信息
key_column_usage	提供关于键列的信息
routines	提供关于过程和函数（但不包括用户定义函数）的信息
views	提供关于所有数据库里的视图的信息
triggers	提供关于所有数据库里的触发器的信息

磁盘空间占用量属于一种元数据，而我将演示如何把它添加到MySQL服务器的信息模式机制里去。因为不需要修改sql_yacc.yy代码或词法散列表，所以整个过程相当简明：在table.h头文件里添加一个助记符，在负责创建信息模式的prepare_schema_tables()函数里，为我们的磁盘占用量查询函数添加一个case分支，定义一个结构来容纳那个新信息模式的列，再把磁盘占用量查询函数的源代码添加到MySQL源代码里。

我们就从把新助记符添加到有关的头文件里开始吧。打开table.h文件，找到enum_schema_tables枚举集合，把新助记符SCH_DISKUSAGE添加到这份清单里。代码清单8-28给出了添加新助记符后的代码片段。

代码清单8-28 修改enum_schema_tables枚举集合

```
enum enum_schema_tables
{
    SCH_CHARSETS= 0,
    SCH_COLLATIONS,
    SCH_COLLATION_CHARACTER_SET_APPLICABILITY,
    SCH_COLUMNS,
    SCH_COLUMN_PRIVILEGES,
    SCH_ENGINES,
    /* BEGIN CAB MODIFICATION */
    /* Reason for Modification: */
    /* This section adds the code to call the new SHOW DISK_USAGE command. */
    SCH_DISKUSAGE,
    /* END CAB MODIFICATION */
    SCH_EVENTS,
    ...
}
```

接下来，需要在负责创建信息模式的prepare_schema_tables()函数里为switch命令增加一个case分支。打开sql_parse.cc文件，把代码清单8-29里的case语句添加进去。请注意，新增的case分支不带break语句，系统将继续检查后面的case分支而不是退出它所在的switch语句。这个技巧可以用来代替

一个冗长的if-then-else-if语句。

代码清单8-29 修改prepare_schema_tables()函数

```
int prepare_schema_table(THD *thd, LEX *lex, Table_ident *table_ident,
                        enum enum_schema_tables schema_table_idx)
{
    DEBUG_ENTER("prepare_schema_table");
    ...
    case SCH_ENGINES:
    /* BEGIN CAB MODIFICATION */
    /* Reason for Modification: */
    /* This section adds the code to call the new SHOW DISK_USAGE command. */
    case SCH_DISKUSAGE:
    /* END CAB MODIFICATION */
    case SCH_COLLATIONS:
    ...
}
```

读者可能已经注意到了，我把新的信息模式命名为DISKUSAGE，是因为DISK_USAGE记号已经在解析器和词法散列表里被定义了一次。如果把新的信息模式命名为DISK_USAGE，在执行SELECT* FROM DISK_USAGE命令时就会遇到错误；因为解析器是把DISK_USAGE记号与SHOW命令而不是与SELECT命令关联在一起的。

接下来，需要定义一个结构来容纳那个新信息模式的列。打开sql_show.cc文件，把一个如代码清单8-30所示的ST_FIELD_INFO类型的新数组添加进去。请注意，在这个新数组里给出的列的名字和类型，与show_disk_usage_command()函数里的有关变量是一致的。

代码清单8-30 DISKUSAGE信息模式的字段信息结构

```
/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SHOW DISK_USAGE command. */
ST_FIELD_INFO disk_usage_fields_info[]=
{
    {"DATABASE", 40, MYSQL_TYPE_STRING, 0, 0, "Database"},
    {"SIZE (Kb)", 21, MYSQL_TYPE_LONG, 0, 0, "Size (Kb)"},
    {0, 0, MYSQL_TYPE_STRING, 0, 0, 0}
};
/* END CAB MODIFICATION */
```

我们还需要给schema_tables数组（也在sql_show.cc文件里）添加一个元素，找到这个数组并把代码清单8-31里的语句添加进去。这条语句将使有关的信息模式函数把表名字DISKUSAGE与字段结构disk_usage_fields_info关联起来，调用create_schema_table()函数来创建新的信息模式，并调用fill_disk_usage()函数来填充新信息模式里的行。那个make_old_format()调用的作用是确保列的名字会显示在屏幕上。最后4个参数依次是：一个函数指针（该函数用来对新信息模式做进一步处理）、两个索引字段、一个bool变量（用来表明这不是不是一个隐藏表）。具体到这个例子，我把那个函数指针设置为NULL（即0，不需要做进一步处理），把那两个索引字段设置为-1（不使用索引字段），把那个bool变量设置为0（不隐藏这个信息模式）。

代码清单8-31 修改schema_tables数组

```

ST_SCHEMA_TABLE schema_tables[] =
{
    ...
    {"ENGINES", engines_fields_info, create_schema_table,
     fill_schema_engines, make_old_format, 0, -1, -1, 0},
    /* BEGIN CAB MODIFICATION */
    /* Reason for Modification: */
    /* This section adds the code to call the new SHOW DISK_USAGE command. */
    {"DISKUSAGE", disk_usage_fields_info, create_schema_table,
     fill_disk_usage, make_old_format, 0, -1, -1, 0},
    /* END CAB MODIFICATION */
    {"EVENTS", events_fields_info, create_schema_table,
     fill_schema_events, make_old_format, 0, -1, -1, 0},
    ...

```

成功在望，剩下的事就是实现fill_disk_usage()函数了。翻到schema_tables数组的前面^①，按照代码清单8-32把fill_disk_usage()函数的代码实现插入到源代码里。

代码清单8-32 添加fill_disk_usage()函数的代码实现

```

/* BEGIN CAB MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SHOW DISK_USAGE command. */
int fill_disk_usage(THD *thd, TABLE_LIST *tables, COND *cond)
{
    TABLE *table = tables->table;
    CHARSET_INFO *scs = system_charset_info;
    List<Item> field_list;
    List<char> dbs;
    char *db_name;
    char *path;
    MY_DIR *dirp;
    FILEINFO *file;
    longlong fsizes = 0;
    longlong lsizes = 0;
    Protocol *protocol = thd->protocol;
    DEBUG_ENTER("fill_disk_usage");

    /* get database directories */
    mysql_find_files(thd, &dbs, 0, mysql_data_home, 0, 1);
    list_iterator_fast<char> it_dbs(dbs);
    path = (char *)my_malloc(PATH_MAX, MYF(MY_ZEROFILL));
    dirp = my_dir(mysql_data_home, MYF(MY_WANT_STAT));
    fsizes = 0;
    for (int i = 0; i < (int)dirp->number_off_files; i++)
    {

```

① 还记得吗？如果没有对某个函数做出声明，就必须让这个函数的代码出现在调用这个函数的语句之前。


```

file = dirp->dir_entry + i;
if (strncasecmp(file->name, "ibdata", 6) == 0)
    fsizes = fsizes + file->mystat->st_size;
else if (strncasecmp(file->name, "ib", 2) == 0)
    lsizes = lsizes + file->mystat->st_size;
}

/* send InnoDB data to client */
table->field[0]->store("InnoDB TableSpace", strlen("InnoDB TableSpace"), scs);
table->field[1]->store((longlong)fsizes, TRUE);
if (schema_table_store_record(thd, table))
    DBUG_RETURN(1);
table->field[0]->store("InnoDB Logs", strlen("InnoDB Logs"), scs);
table->field[1]->store((longlong)lsizes, TRUE);
if (schema_table_store_record(thd, table))
    DBUG_RETURN(1);

/* now send database name and sizes of the databases */
while (db_name = it_dbs++)
{
    fsizes = 0;
    strcpy(path, mysql_data_home);
    strcat(path, "/");
    strcat(path, db_name);
    dirp = my_dir(path, MYF(MY_WANT_STAT));
    for (int i = 0; i < (int)dirp->number_of_files; i++)
    {
        file = dirp->dir_entry + i;
        fsizes = fsizes + file->mystat->st_size;
    }
    restore_record(table, s->default_values);
    table->field[0]->store(db_name, strlen(db_name), scs);
    table->field[1]->store((longlong)fsizes, TRUE);
    if (schema_table_store_record(thd, table))
        DBUG_RETURN(1);
}

/* free memory */
my_free((gptr)path, MYF(0));
DBG_RETURN(0);
}
/* END CAB MODIFICATION */

```

注解 如果你使用的是Windows平台，请把my_malloc()调用里的PATH_MAX替换为MAX_PATH，再把strncasecmp调用全部替换为strnicmp调用。

我从上一节SHOW DISK_USAGE命令的例子复制了一些代码，删除了用来创建字段的调用（这部分工作现在由disk_usage_fields_info数组负责完成）和向客户端发送行的代码。我这次使用了TABLE类/结构的一个实例来保存fields数组里的值，因为数组的下标从零开始计数，所以fields[0]对应着第一个列。

现在，编译MySQL服务器的准备工作已全部完成。因为你修改了一个关键的头文件（table.h），所以这次编译所花费的时间可能会比平时长一些。如果遇到编译错误，请改正它们之后再往下继续。

在编译完MySQL服务器并得到一个新的可执行文件后，关闭你正在使用的MySQL服务器，把新的可执行文件复制到MySQL的安装目录，然后重新启动MySQL服务器。现在，你就可以在一个MySQL客户端工具里试试新的信息模式了。代码清单8-33给出了试用新的DISKUSAGE信息模式的一个例子。

代码清单8-33 试用DISKUSAGE信息模式

```
mysql> USE INFORMATION_SCHEMA;
Database changed
mysql> SHOW TABLES;
```

```
+-----+
| Tables_in_information_schema |
+-----+
| CHARACTER_SETS              |
| COLLATIONS                   |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                     |
| COLUMN_PRIVILEGES            |
| ENGINES                      |
| DISKUSAGE                    |
| EVENTS                       |
| FILES                         |
| KEY_COLUMN_USAGE             |
| PARTITIONS                   |
| PLUGINS                      |
| PROCESSLIST                  |
| ROUTINES                     |
| SCHEMATA                     |
| SCHEMA_PRIVILEGES            |
| STATISTICS                   |
| TABLES                      |
| TABLE_CONSTRAINTS           |
| TABLE_PRIVILEGES            |
| TRIGGERS                     |
| USER_PRIVILEGES              |
| VIEWS                        |
+-----+
23 rows in set (0.00 sec)
```

```
mysql> SELECT * FROM DISKUSAGE;
```

```
+-----+-----+
| DATABASE | SIZE (Kb) |
+-----+-----+
| InnoDB TableSpace | 10485760 |
| InnoDB Logs        | 20971520 |
| cluster            | 9867      |
| mysql              | 617310    |
+-----+-----+
```

```
| test          | 9720      |  
+-----+  
5 rows in set (0.31 sec)
```

既然你已经知道了如何添加一个新的信息模式，就可以为那些数据库管理员们提供更直接、更细致的MySQL服务器管理工具了。

8.5 小结

本章演示了如何通过添加新函数和新命令来扩展MySQL服务器功能。

本章讲解了如何创建一个能够动态加载和卸载的UDF库，如何在MySQL服务器的源代码里添加一个新的本机函数，以及如何在MySQL解析器和查询执行代码里添加一条新的SHOW命令。文中还讲解了如何为MySQL服务器添加一个新的信息模式。

多种多样的功能扩展手段使得MySQL服务器比其他任何一种数据库系统都更加灵活。UDF机制是最容易通过编程实现的办法之一，而MySQL的UDF机制无论是在复杂性方面还是在速度方面都把竞争对手抛在了后面。MySQL服务器是一个开源软件的事实意味着我们还可以通过修改它的源代码和添加自己的SQL命令来满足我们的需要。无论你怎么使用这些工具，相信你会为自己有能力摆脱“现成的”函数和命令的束缚而感到自由和骄傲。

下一章将讨论一些数据库服务器设计和实现方面的高级主题，通过下一章的学习，可以为使用MySQL服务器的源代码作为研究数据库系统内部组成的实验平台做好准备。



Part 3

第三部分

高级数据库的内部组成

这一部分将深入MySQL系统的内部去探查到底是什么让这个系统工作的。第9章将对MySQL体系结构中的查询执行做进一步的讨论并演示如何进行各种源代码实验。第10章将讨论MySQL内部查询表示并提供一个可以替代MySQL内部查询表示的示例查询表示。第11章将探讨MySQL内部查询优化器，并描述一个可以替代MySQL内部查询优化器的示例查询优化器，它使用的是第10章实现的内部表示。第11章还讲述如何通过修改MySQL源代码来实现一个新的查询优化器。第12章把前几章介绍的技术结合起来去实现一个新的查询处理引擎。

本 部 分 内 容

- 第 9 章 数据库系统的内部组成
- 第 10 章 内部查询表示
- 第 11 章 查询优化
- 第 12 章 查询执行

本章重点介绍一些数据库系统内部的概念，为后面几章深入分析数据库系统的内部工作情况打下基础。本章从一个更深的层次对查询命令在服务器内部的表示以及查询命令在服务器内部的执行过程进行剖析。首先从一个比较广泛的视角去分析这些问题，然后再以MySQL为例，讨论如何具体使用MySQL系统来进行软件工程实验。最后，把几个数据库系统内部组成的实验项目留作练习。

9.1 查询执行

绝大多数数据库系统要么使用迭代型执行策略，要么使用解释型执行策略。迭代型方法产生完成具体操作（联结、投影等）所要用到的一系列调用，但并不是用来并入内部表示的各个功能的。把一条查询转换成一个方法调用序列的工作，需要用到各种功能编程技术和程序转换技术。有好几种算法可以从基于关系代数的查询说明生成迭代型程序。

查询执行机制的实现创建了一组已定义的已编译功能指令，这些指令是用某种高级程序设计语言编写出来的，再通过一个调用栈或一个过程调用序列把它们链接在一起。当创建了查询执行计划并将其选中以备执行的时候，会有一个编译器（通常就是用来创建这个数据库系统的那个）把相应的过程调用序列编译成一段可执行的二进制代码。因为编译型执行策略的开销比较大（需要进行编译），所以这类数据库系统通常会把已编译的执行计划保存起来，以便在今后遇到类似或同样的查询时可以重复使用它们。

另一方面，解释型（interpretative）方法采取了使用基本操作的已编译抽象来完成查询执行。选定的查询执行计划被重新构造成一个方法调用队列，这个队列里的方法将依次从队列取出接受处理，上一个方法的执行结果（输出）留在内存里作为后续方法的输入。这种策略的具体实现通常被称为惰性求值，这是因为可用的已编译方法虽然也经过优化，但这种优化往往侧重于总体平衡，而不是最优性能。

9.1.1 重温 MySQL 查询执行

MySQL在处理和执行查询命令时采用的是解释型策略。它被实现为一个线程化的体系结构，每个查询命令都有它自己的执行线程。图9-1给出了MySQL查询处理方法的构成情况。

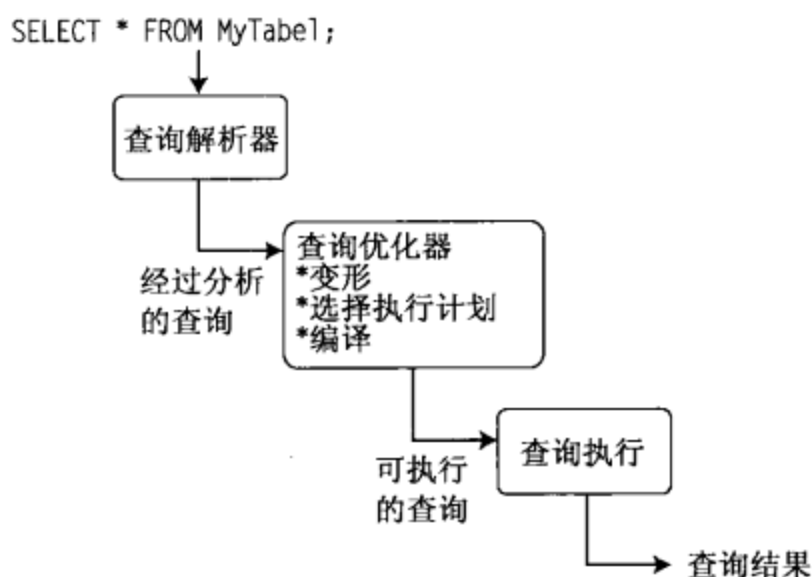


图9-1 MySQL的查询执行

当客户端发出一条查询命令时，MySQL服务器将创建一个新的线程并把SQL命令传递到查询解析器进行语法检查。正如你在前一章里看到的那样，MySQL解析器是用一个非常大的Lex-YACC脚本实现之后又用Bison工具编译得到的。解析器将构造一个用来存放查询语句（SQL）的查询结构，这个结构可以用来执行查询。把查询结构创建出来以后，控制权将转到查询处理器（query processor）那里，在那里进行一系列检查，比如检查表的完整性，检查用户是否有足够的操作权限等。如果用户的操作权限没有问题，也顺利地打开了表（如果是UPDATE操作，还需要锁定表），控制权将转到那些用来完成读取数据、筛选数据、排除无关数据等基本查询操作的方法。优化工作是通过编排各有关表和各有关操作的顺序在那个代表查询语句的数据结构上进行的，优化器将根据一系列规则生成一个更有效率的查询执行计划。这种优化策略被称为“选取-投影-联结”（SELECT-PROJECT-JOIN）策略。最后，查询操作的结果将由一些网络通信例程（通信协议）和数据读写例程（访问方法）返回给客户端。

9.1.2 什么是已编译查询

“已编译”是个经常会引起一些误会的概念。有些人认为，已编译查询（compiled query）是一个真正编译出来的迭代查询执行计划。但有些研究人员（比如C.J.Date）认为，只要一个查询执行计划已经接受了优化并被保存起来供以后执行，它就是一个已编译查询了。因为存在着这两种看法，我们在使用已编译查询的时候，应该多加小心。本书一直避免使用已编译这个词，因为MySQL的查询优化器和查询执行引擎都不会把查询执行计划保存起来供以后重复使用，而查询执行也不需要经过任何编译或汇编便能正常工作。

注解 已存储过程（stored procedure）的定义是已保存的计划——它是已编译或经过优化的，供日后执行，只要你提供的数据能满足它对输入参数的要求，就可以运行任意多次。

9.2 深入MySQL的内部

如果不让某人参与一个项目，你怎么向他解释优化器是如何工作的？如果不深入一个数据库系统的内部，你又怎么了解它的内部工作情况？有些东西单靠书本是教不好也学不会的。本节将讨论如何

使用MySQL作为一个实验平台来培训职业程序员或进行学术研究。

9.2.1 开始用 MySQL 做实验

用MySQL做实验的方式有好几种。比如说,你可以使用一个交互式调试器来观察其内部组件的工作情况,也可以使用MySQL系统作为一个宿主系统来检验你为实现某种高级数据库技术而编写的代码。

如果你打算进行这样的实验,请考虑搭建一个专用的实验平台。这很有必要,尤其是打算对MySQL进行某种扩展的时候。不要在用来开发项目的服务器上冒险,你的开发成果说不定会被实验毁于一旦。

1. 用MySQL源代码做实验

如果你想研究MySQL服务器本身,最具侵入性的实验方式莫过于修改它的源代码了。这类实验通常是先观察系统的运行情况,再设计一些实验来修改它的某个部分(换用另一种算法,替换一段代码等),然后再观察系统的行为有什么变化。这个办法的好处是你迟早会摸清MySQL的底细,缺点是像这样修改它的源代码往往会导致服务器不稳定或根本无法继续使用——在换用另一种算法或另一种数据结构的时候,最容易发生这样的事情。可话又说回来了,如果你真的想精通MySQL源代码,没有什么办法比在实践中进行观察更好的了。通过这类实验收集到的信息还可以用来设计其他类型的实验。

2. 把MySQL作为宿主系统来实验新技术

把MySQL作为宿主系统来检查实验代码是一种比较平和的实验手段。你可以把实验重点放在(比如说)查询优化器或查询执行引擎身上,而无需担心系统的其他部分。任何一个数据库系统都是由许许多多的部分组成的。就拿MySQL来说,除了网络通信、数据输入和访问控制等几个比较大的子系统,它还包括了许多用来管理文件和内存的工具。与其自行创建这些子系统,还不如把有关的MySQL源代码复制到自己的代码里来使用。

本书里的实验项目都是采用这个办法实现的。我将演示如何连接到MySQL解析器,以及如何使用MySQL解析器来读取、测试、接受合法的命令并让代码转向实验项目的优化器和执行例程。

解析器和词法分析器的工作是识别那些在解析器或词法散列表里定义的由字母和数字构成的字符串(它们统称为“记号”)。解析器会给所有的记号加上一个位置信息(在输入流里出现的顺序),它还会使用识别非记号字符串的特定模式的逻辑来识别字面量和数值。解析器的工作结束后,控制权将转到词法分析器。MySQL里的词法分析器可以识别出记号和非记号的特定模式。一旦识别出了合法的命令,控制权将被转给每条命令的执行代码。通过修改MySQL解析器和词法分析器,我们就可以在实验项目里识别新的记号或关键字了。修改解析器和词法分析器的具体步骤请参见本书第8章。我们可以设计一些新的命令来模仿甚至取代各种SQL命令,无论是用来处理数据的select、update、insert和delete命令,还是用来定义数据的create和drop命令,都可以成为我们实验的对象。

在控制权转到被实验的优化/执行引擎后,我们就可以进行各种实验了。比如说,可以把MySQL的内部查询表示结构转换为另一种结构,让我们自己的查询优化器和查询执行引擎使用这种新结构去执行查询,最后通过使用MySQL系统把结果集发送给客户端的办法把结果返回给客户端。这么做的好处是,我们可以把精力集中在自己的内部数据库组件上,通用的网络通信组件和词法/语法分析组件就用MySQL的好了。

3. 运行MySQL的多个实例

很少有人知道在同一台机器上可以同时运行多个MySQL服务器的实例。换句话说，你完全可以让修改后的MySQL系统与日常使用的开发系统运行在同一台机器上。如果你的资源有限，或者想对修改前后的MySQL服务器进行对比，这个办法值得考虑。在同一台机器上运行多个MySQL服务器的实例需要在命令行或配置文件里设置特定的参数。

在最简单的情况下，需要为不同的MySQL服务器实例指定一个不同的TCP/IP端口或套接字来实现通信，并为它们分别指定一个子目录来存放数据库文件。下面是在一台Windows机器上启动第二个MySQL实例的命令示例。

```
mysqld-debug --port 3307 --datadir="c:/mysql/test_data" --console
```

这个例子让第二个服务器实例使用TCP 3307号端口（默认端口是3306号），使用另一个数据目录，并运行为一个控制台应用程序。如果想连接第二个服务器实例，就必须告诉客户端程序使用3307号端口。下面是用来连接第二个服务器实例的命令。

```
mysql -uroot --port 3307
```

mysqladmin工具也支持--port参数。比如说，如果想关闭运行在3307号端口上的第二个服务器实例，可以发出如下所示的命令。

```
mysqladmin -uroot --port 3307 shutdown
```

如果说这个技巧有什么不好的地方，那就是很容易让人忘记自己连接的是哪一个实例。为了避免混淆，更为了避免把一条后果严重的查询命令（比如DELETE或DROP）错发给一个服务器，可以用下面这个技巧给自己加个保险：修改MySQL客户端工具的命令提示符，让它告诉你正连接在哪一台服务器上。比如说，用prompt DBXP->命令来设置连接着实验服务器的MySQL客户端程序的命令提示符，用prompt Development->命令来设置连接着开发服务器的MySQL客户端程序的命令提示符。这个技巧可以让你一眼看出是在向哪个服务器发出命令。代码清单9-1给出了一个在MySQL客户端程序里发出prompt命令的例子。

代码清单9-1 用prompt命令改变MySQL客户端程序的命令提示符

```
mysql> prompt DBXP->
PROMPT set to 'DBXP->'
DBXP->show databases;
```

```
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| test |
+-----+
3 rows in set (0.01 sec)
```

```
DBXP->
```

提示 还可以用\ d选项让当前数据库的名字也出现在命令提示符里。比如说,如果你在连接着实验服务器的客户端程序里发出prompt DBXP:\ d->命令,它的命令提示符就会变成DBXP:Test->的样子,其中DBXP是实验服务器的名字,Test是当前数据库的名字(会随use命令而变化)。

在Linux平台上也可以做同样的事情。或者,可以使用一个名为mysqld-multi的脚本工具来配置多个MySQL服务器实例(这个脚本在Windows平台上不能用)。这个脚本会在设置好正确的参数后调用mysqld_safe或mysqld脚本来启动各个服务器。为了管理多个服务器实例,可以使用MySQL Instance Manager工具(mysqlmanager,这个工具也只能在Linux平台上使用)来监控那些服务器和它们的状态。你可以使用这个工具在本地或远程对服务器进行管理。

可以使用这个技巧来限制其他人对修改后的服务器进行访问。如果改变了端口号或套接字,就只能是那些知道正确参数的人才能与服务器建立连接。这可以减少把你的修改向所有用户公开的风险。如果在你的开发环境里有许多目的各异的实验和研究项目,并且它们还共享着同样的资源(这在校园里很常见),可以采取这些步骤来保护自己的实验项目不会与其他项目相互影响。这通常不是什么大问题,但采取点儿预防措施总比没有措施强。

注意 如果启用了二进制查询日志、查询日志或慢查询日志,还需要为MySQL服务器的每一个实例分别指定一个日志文件的存放地点。不这样做有可能导致日志文件和/或数据损坏。

9.2.2 注意事项

说到用MySQL来做实验,最大的挑战应该是为了让解析器能够识别出SQL命令里的新关键字而对它进行修改(请参见第8章)。YACC语言不算复杂,也不是一种刚出现不久的新语言,但修改YACC文件需要极大的耐心和细心。必须把新命令的MySQL语法定义放在每一个解析器命令定义的前面,不能遗漏任何一处。只有这样,才能拦截到解析器的执行流并把它转到自己的查询引擎。

最频繁发生和最没有难度的挑战是跟上MySQL源代码的升级脚步。坏消息是这种升级的频率无法预测。如果你想使用最新的功能,就不得不在MySQL AB公司每推出一个新版本时,把你的修改插入到最新的MySQL源代码中,这也许并不复杂,但能长期坚持下来的人恐怕不多。如果你正在编写的扩展需要紧跟MySQL推出升级版本的脚步,你可以考虑这么做:再建立一个服务器来做实验,开发工作仍在你的“老”服务器上进行。

提示 最可能遇到的挑战是在MySQL的代码库中发现各种内部数据表示的含义、布局和使用方法。解决这个问题的办法只有一个:熟读MySQL的源代码。建议读者访问并阅读MySQL网站、博客和论坛上的文档(《MySQL参考手册》和MySQL System Internals Manual)以及各种文章。它们是宝贵的信息财富。有些文章里的概念可能不那么容易消化,需要多看几遍才能理解。但不要因为看不懂文档而烦恼,休息一下,过些时候再去阅读它们。我每次阅读那些技术资料都会有新的收获。

9.3 数据库系统内部组成实验

我创建了一个数据库实验项目(DBXP),来帮助大家探索MySQL的内部世界和实验各种数据库

组件的替代实现方案。可以通过这个项目学到更多关于数据库系统的构造和内部工作原理的知识。

9.3.1 为什么叫实验

DBXP是一个实验项目，不是一个解决方案，因为它并不完整。这主要体现在以下几个方面：我在实现各有关技术时把出错处理部分压缩到了最低限度，它的功能不够丰富，它的健壮性比较低。这并不意味着DBXP里的技术不能在经过完善后取代MySQL系统现有的某些组件；要知道，DBXP是一个研究和探索性的项目，而不是一个产品开发项目。

9.3.2 实验项目概述

DBXP是由一系列采用另外一种算法和机制实现的类所构成的项目，这些类实现了内部查询表示、查询优化、查询执行和文件访问。这些类不仅给了你们一个学习查询优化理论并亲身探索这些理论的高级实现的机会，还使DBXP技术的核心成果可以在无需修改MySQL内部操作的情况下运行。因为新增加的DBXP技术没有影响到MySQL现有的内核代码，所以可以放心大胆地进行各种相关实验。这种安全性有助于降低对一个现有系统进行修改而带来的风险。

MySQL解析器的实现（见sql_parse.cc文件）通过调用为每个SQL命令所实现的函数来把控制权转交给执行子进程的指定实例。比如说，SHOW命令将被转给在sql_show.cc文件里实现的函数去处理。因此，为了把处理工作转到DBXP查询处理器中进行，就需要对sql_parse.cc文件里的MySQL解析器代码进行修改。

DBXP查询处理器里的第一步是把MySQL的内部表示转换为我们所实验的内部表示。所选用的内部表示叫作查询树（query tree），这棵树的每个结点包含着一个原子化关系操作（选取、投影、联结等），结点之间的接线代表着数据流和流向。图9-2给出了查询树的一个概念性例子。这个例子使用了以下记号：投影/选取（ Π ）、限制（ Σ ）、联结（ Φ ）。图中的箭头代表着数据从表到根结点的流动方向。联结操作表示为一个有两个子结点的结点，来自子结点的数据在经过联结操作的处理后，被传递到它的上一级结点（父结点）。每个结点可以有零个、一个或两个子结点，但有且只有一个父结点。

之所以选用查询树，因为它可以让DBXP查询优化器使用各种基于树结构的算法。换句话说，DBXP优化器将使用树结构和树处理算法把各个结点在树里的位置调整为一种更有执行效率的顺序。不仅如此，经过优化的查询是通过遍历查询树的叶结点而得到执行的，在每一个结点完成该结点上的操作，再把信息沿着结点之间的连线传递到上一级。这个技术还使得数据在查询执行过程中能够以一种流水线的方式传递：数据从叶结点开始向根结点传递，每次传递一个数据项。

为一个数据项向下遍历到叶结点再向上返回到树[这个过程称为脉动（pulsing）]，可以让每个结点处理一个数据项，每次返回结果集里的一行。查询树的这种脉动使得执行过程呈流水线形式，其结果是可以更快地返回最初几条查询结果，这意味着客户端在更短的时间里就可以收到查询结果。看到查询结果这么快就呈现在眼前，虽然不是全部，但会让用户感到这是一个速度很快的系统。

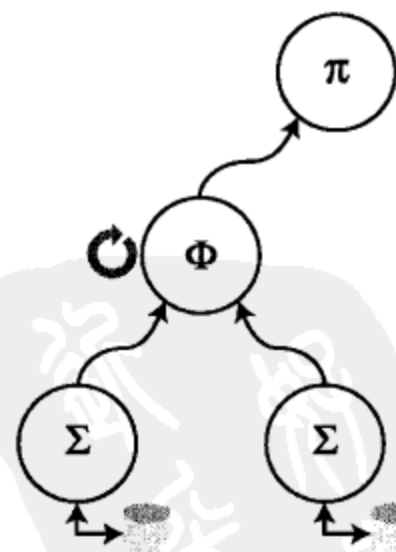


图9-2 查询树概念

以MySQL作为宿主系统的DBXP项目最终实现了这样一个效果：先由MySQL解析器对查询命令进行分析，然后由DBXP代码负责完成查询的优化和执行，最后由MySQL的网络通信例程以每次一行的方式，把查询结果返回给客户端。

9.3.3 实验项目的组成部分

我设计这个实验项目的目的是为了让你了解数据库系统的核心组件还可以有其他的实现方式，并允许你通过为这个项目添加自己的修改来探索这些实现。DBXP是用一组简单的C++类实现的，它们代表着数据库系统里的各种对象。

元组、关系、索引和查询树都有与之对应的类。为了管理多用户对表的访问，我还增加了一些辅助性的类。图9-3是DBXP的高级架构示例。

表9-1列出了这个项目用到的类。这些类分别存放在与它们同名的源文件中。比如说，Attribute类的定义存放在attribute.h文件里，它的代码实现存放在attribute.cc文件里。

表9-1 DBXP项目里使用的类

类	说 明
Query_tree	提供了查询命令的内部表示，还包含着查询优化器
Expression	提供了一种表达式求值机制
Attribute	封装对元组（行）的属性（列）进行存储和处理的各种操作

这些类代表着一个数据库系统的基本构件。第10~12章将分别对查询树、启发式优化器和流水线执行算法做专题讨论，还会把一些工具软件介绍给大家。我将向读者演示DBXP项目一部分组件（最复杂的那些）的实现细节，剩下来的一部分留给大家作为练习。

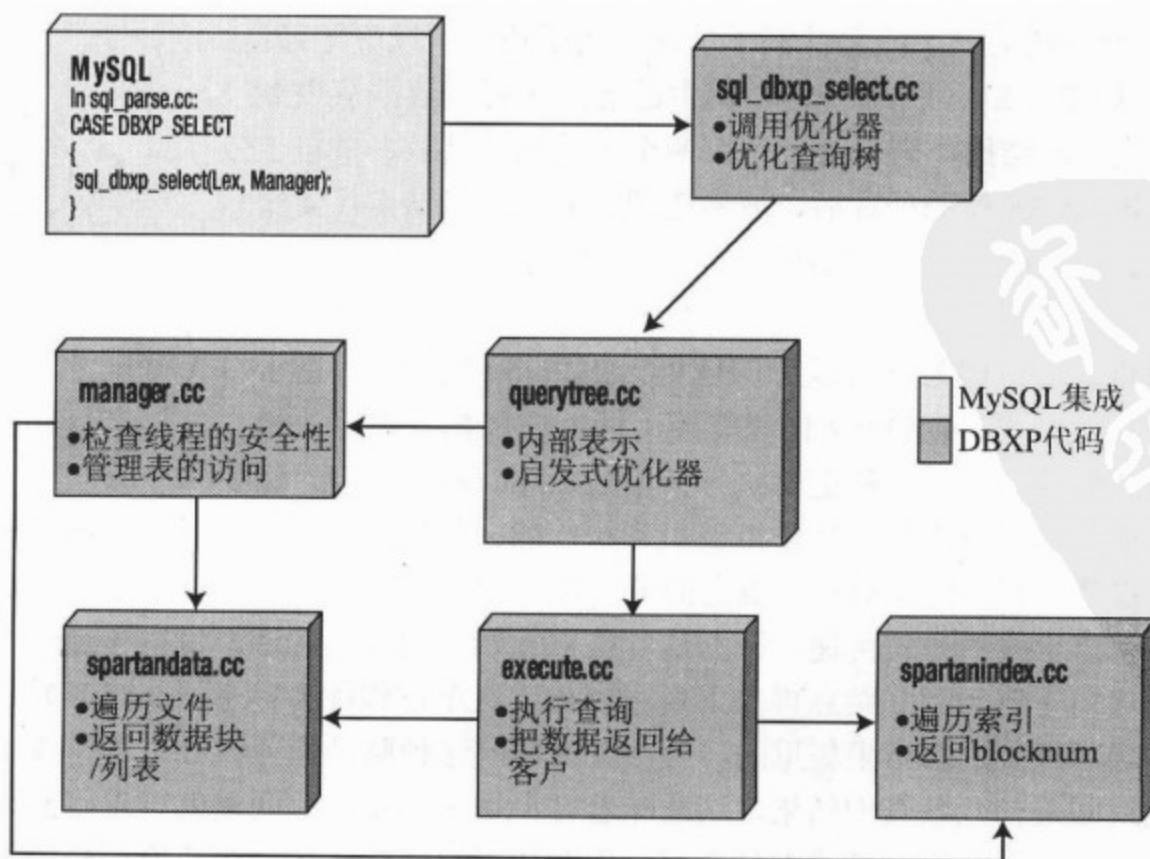


图9-3 实验项目的高级框图

注解 我在本书的前言部分对如何在课堂教学环境里使用这个实验项目提出了一些建议。

9.3.4 在 Linux 平台上进行实验

在Linux平台上运行这个实验需要为这个新项目创建一些制作文件并把它们随MySQL服务器一起编译。这些项目文件不需要任何特殊的编译和库。对MySQL配置文件和制作文件进行修改的细节，将在下一章演示如何为这个实验项目构造“假”SQL命令的时候讨论。

如果你既没有试过此前各章里的示例程序，也没有在Linux工作站上编译过它们，请看我列在这里的MySQL制作文件的基本修改步骤。下面是为DBXP项目修改有关文件的基本过程。

- (1) 在MySQL源代码根目录下创建一个名为DBXP的子目录。
- (2) 在DBXP项目目录里创建一个Makefile.am文件。
- (3) 把这个项目添加到configure和configure.in文件里去。
- (4) 运行./configure、make和make install命令。
- (5) 关闭MySQL服务器，把编译出来的可执行文件复制到MySQL服务器的binary子目录。
- (6) 重新启动MySQL服务器，用MySQL客户端工具连接它，运行DBXP SQL命令。

9.3.5 在 Windows 平台上进行实验

在Windows平台上运行这个实验，需要在MySQL服务器的解决方案(mysql.sln)里创建一个新的项目文件。对解决方案进行修改的细节，将在下一章演示如何为这个实验项目构造“假”SQL命令的时候讨论。

如果你既没有试过此前各章里的示例程序，也没有在Windows系统上编译过它们，请看下面所列的MySQL制作文件的基本修改步骤。以下是为DBXP项目修改有关文件的基本过程。

- (1) 在MySQL源代码根目录下创建一个名为DBXP的子目录。
- (2) 在Visual Studio (2003或2005)里打开mysql.sln解决方案文件。
- (3) 在DBXP项目目录里创建一个新的Win32项目。
- (4) 编译MySQL服务器。
- (5) 关闭MySQL服务器，把编译出来的可执行文件复制到MySQL服务器的binary子目录。
- (6) 重新启动MySQL服务器，用MySQL客户端工具连接它，运行DBXP SQL命令。

9.4 小结

本章简要介绍了几种比较复杂的数据库核心技术。主要讲解了查询在服务器的内部是如何表示的，又是如何执行的。最重要的是，阐述了应该如何使用MySQL去进行自己的数据库实验。了解了这些技术可以帮助你进一步理解为什么和怎样构建MySQL服务器，以及它是如何执行的。

下一章将通过一个具体的例子对如何利用查询树结构来实现内部查询表示做专题讨论。第10~12章的内容将讲解如何实现查询优化器和查询执行引擎。如果你想了解究竟怎样才能构建一个数据库系统，下一章就将介绍如何创建你自己的查询引擎。

本章是数据库实验项目 (DBXP) 将要实现的高级数据库技术中的第一部分。首先介绍一些与用来在内存里存储查询的查询树结构有关的概念, 然后给出这个项目所使用的查询树结构并开始实现第一批 DBXP 代码。本章最后留了几个练习, 通过这些练习你可以学到更多关于 MySQL 和查询树的知识。

10.1 查询树

查询树是一种对应于查询的树结构, 这种树的叶结点包含访问一个关系结点和带有零个、一个或多个子结点的内部结点。内部结点包含关系操作符。这些操作符包括投影 (用 π 表示)、限制 (用 σ 表示) 和联结 (用 θ 或 \bowtie 表示)^①。树中结点之间的连线代表数据从下向上的流动方向——从叶结点 (对应着数据库里的数据) 到根结点 (用来产生查询结果集的最后一个操作符)。图 10-1 给出了一个查询树的例子。

查询树的求值规则是: 只要某个内部结点的操作数都进行了求值, 就对该结点进行求值, 并把求值结果传递给它的父结点; 求值过程将在对根结点进行了求值并把它替换为构成查询结果的元组时结束。接下来的几节描述了查询树结构的一种变体, 我将使用这种变体把查询的表示保存在内存里。与使用一种基于关系演算的内部表示相比, 使用这种机制有许多好处, 表 10-1 对此进行了汇总。

```
SELECT name
FROM faculty f, classes c
WHERE f.id = c.fac_id AND
f.department_id = 'CS' AND c.semester = 'F2001'

name( $\sigma_{\text{department\_id}='CS' \wedge \text{semester}='F2001'}(\pi_{\text{id}=\text{fac\_id}} c)$ )
```

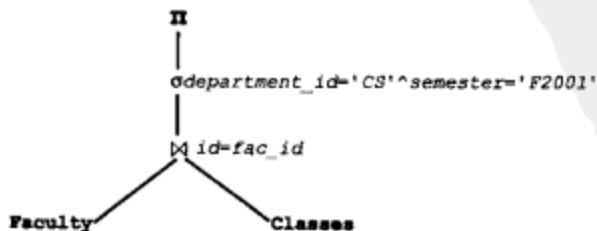


图 10-1 查询树的例子^②

- ① 很奇怪, 很少有书解释这些符号的来历。从传统上讲, θ 代表条件联结 (θ 联结), \bowtie 代表自然联结, 但大部分书刊都不区分这些概念, 并用一种符号 (或两种符号混着用) 来代表所有的联结。
- ② 你还会在本书的其他地方看到与此类似的图。在这个图里有一个与数据库理论相违背并经常为人们所忽视的细节。你能看出来吗? 提示: semester 属性的域是什么? 对列里的数据进行编码违反了哪条原则?

表10-1 使用查询树与关系演算的好处

关系操作的要求	查 询 树	关系演算
能否减少	能。在开始对查询计划进行求值之前可以对查询树进行删减	只能通过应用代数运算符来实现
能否用来执行查询	能。查询树可以用来执行查询。通过把数据从叶结点向上传递到根结点的办法就可以做到这一点	不能。需要转换为另一种形式
能否支持关系代数表达式	能。查询树对关系代数的支持非常好	不能。需要转换
能否在数据库系统中实现	能。树结构是一种常用的数据结构	只能通过对演算建模的各种设计来实现
能否包含数据	能。树结点可以包含数据、操作和表达式	不能。只能包含构成表达式的字面量和变量

显然，查询树内部表示要优于现代数据库系统使用的传统机制。比如，MySQL里的内部表示由一组为了简化（或者说加快）遍历操作而专门设计的类和结构组成，它们包含着查询和它的元素。它为查询优化器和查询执行管理着数据^①。

查询树内部表示也有一些不足。大多数优化器都无法对树结构进行处理。如果能让优化器支持查询树，就需要修改这个优化器。类似的，支持查询树的查询执行与大多数查询处理的具体实现有很大的不同。在这种情况下，查询执行引擎将从查询树运行，而不是作为另外一个步骤运行。后面的几章将在讨论如何创建另一个可选的优化器和执行引擎的时候解决这些问题。

DBXP查询树是一个树状的数据结构，它使用了一个结点结构来容纳为了描述以下操作所需要的所有参数。

- 限制，把与某个属性表达式相匹配的结果检索出来。
- 投影，提供了选取将被包括在结果集里的属性的能力。
- 联结，把两个或更多个关系合并在一起，用结果集里的属性生成一个复合集合。
- 排序，对结果集进行排序。
- 存异，剔除重复的元组，让结果集里只有彼此不相同的元组。

注解 存异操作是为了让DBXP能够完成一种大多数SQL实现都不支持的操作而特意增加的，它不是关系代数的固有属性。

投影、限制和联结是最基本的关系操作。排序操作和存异操作是我为了保持查询树的完整性而特意增加的辅助性操作（这样才能让所有可能的操作都可以用查询树里的一个结点来表示）。联结操作可以有联结条件（ θ 联结），也可以没有任何条件。联结操作可以细分为以下几种。

- 内联结，对两个关系进行联结，返回匹配的元组。
- 外联结（左、右、全），至少返回 FROM 子句所列出的某一个表或视图的全部行——只要那些行满足全部的 WHERE 搜索条件。左外联结操作将返回位于联结操作符左侧的那个表的全部行；右外联结操作将返回位于联结操作符右侧的那个表的全部行；全外联结操作将返回位于联结操作符左、右两侧的两个表的全部行。不相匹配的行的属性值将返回为空值。

^① 像MySQL那样用内部表示来为优化器管理数据并不是唯一的办法，还有许多方案可供选择。查询树就是人们为了能够直接对之进行优化和执行而设计出来的一种数据结构。

- 左外联结，对两个关系进行联结，返回匹配的元组和位于联结操作符左侧的那个表的全部元组，来自另一个关系的未匹配属性将全部返回为空值 (null)。
- 右外联结，对两个关系进行联结，返回匹配的元组和位于联结操作符右侧的那个表的全部元组，来自另一个关系的未匹配属性将全部返回为空值 (null)。
- 全外联结，对两个关系进行联结，返回两个关系里的所有元组，来自另一个关系的未匹配属性将全部返回为空值 (null)。
- 叉积，对两个关系进行联结，把第一个关系里的每一个元组映射到另一个关系里的全部元组上。

查询树还支持一些集合操作，其中包括。

- 并，对两个有着同样结构的关系进行联结，只返回满足条件的匹配；相当于数学集合理论中的“并”操作。
- 交，对两个有着同样结构的关系进行联结，只返回不满足条件的匹配；相当于数学集合理论中的“交”操作。

什么是 θ 联结

你们也许正想知道为什么有些联结操作被称为等式联结，另一些则被称为 θ 联结。“等式联结”是联结条件是一个等式(=)的联结操作； θ 联结是联结条件是一个不等式(>、<、>=、<=、<>)的联结操作。从理论上讲，所有的联结操作都是 θ 联结；但在实际工作中， θ 联结相当少见，等式联结则很常见。

DBXP查询树虽然提供了并和交操作，但绝大多数数据库系统实现的并操作都是把两个结果集简单地合二为一。MySQL解析器目前还不支持交操作，但它支持并操作。要想实现交操作，就必须进一步修改MySQL解析器。接下来的几个小节描述了主要的代码实现，以及为了把MySQL的内部查询表示转换为一个DBXP查询树而创建的类。

10.1.1 查询转换

MySQL解析器必须加以修改才能识别和分析DBXP SQL命令。我们需要一个办法来告诉解析器我们想使用DBXP，而不是使用现有的查询引擎。为了简化修改工作，只增加了一个关键字DBXP。在SQL命令里加上这个关键字将使得解析器把MySQL的内部查询表示转换为DBXP内部表示。虽然这个过程需要花费一些时间，并需要进行少量的额外运算，但这个方案大大降低了修改MySQL解析器的工作量，同时还提供了一种把DBXP数据结构与MySQL数据结构进行对比的通用机制。在后面的讨论里，我将把添加了DBXP关键字的SQL命令简称为“DBXP SQL命令”。

把MySQL的内部查询表示转换为DBXP内部表示^①的过程开始于MySQL解析器，它负责识别DBXP SQL命令。在它识别出DBXP SQL命令之后，系统将把控制权转交给一个名为sql_dbxp_parse.cc的类，它将负责把MySQL的内部查询表示转换为DBXP内部表示(查询树)，具体的转换工作将由一

① 虽然许多关于查询处理这一主题的文章并不赞同进程要互相区别开，但却一致认为必须执行某个不同的进程步骤。

个名为build_query_tree()的方法负责完成。这个方法目前只在遇到带有DBXP关键字的SLEECT和EXPLAIN SELECT命令时才会被调用。

10.1.2 DBXP 查询树

DBXP查询优化器的核心是DBXP内部查询表示数据结构,这个数据结构代表着一条经过分析和转换的DBXP SQL命令。

这个结构被实现为一个树状结构(所以被称为查询树),它的每个结点可以有零个、一个或两个子结点。有零个子结点的结点是这个树的叶结点,有一个子结点的内部结点代表着一个单元操作,有两个子结点的内部结点代表着一个联结操作或一个集合操作。代码清单10-1给出了DBXP查询树的结点结构的源代码。

代码清单10-1 DBXP查询树的结点

```
struct query_node
{
    query_node();
    ~query_node();
    int          nodeid;
    int          parent_nodeid;
    bool         sub_query;
    bool         child;
    query_node_type node_type;
    type_join    join_type;
    join_con_type join_cond;
    COND         *where_expr;
    COND         *join_expr;
    TABLE_LIST  *relations[4];
    bool         preempt_pipeline;
    List<Item>    *fields;
    query_node    *left;
    query_node    *right;
};
```

在DBXP查询树的结点结构里,有些变量用来管理结点的排列情况和构成DBXP查询树本身。nodeid和parent_nodeid变量用来建立结点之间的父子关系。这很有必要,因为优化器在进行优化时会把结点在查询树上四处挪动。使用parent_nodeid变量可以避免在查询树里使用逆向指针(reverse pointer)^①。

sub_query变量用来标识一个子查询的起始结点。这意味着这种数据结构可以支持嵌套查询(子查询),而无需对其结构进行其他修改。需要说明的是,DBXP优化器使用的优化算法把子查询标志(sub_queries变量)视为树遍历操作的一个停止条件。一旦遇到这个标志是TRUE的结点,查询优化例程就将以该结点作为起点重新运行,开始一次新的优化。这意味着DBXP查询树可以支持任意多个子查询并把它们表示为查询树里的子树;这是查询树的一个重要优点,许多其他类型的内部查询表示都做不到这一点。

① Knuth和其他一些算法专家强烈反对在树结构里使用逆向指针。

where_expr变量是一个指向MySQL中的COND结构的指针，该结构管理着一个典型的通用表达式树。

relations数组用来存放指向各种relation（关系）类的指针，那些relation类代表着MySQL存储引擎里使用的各种内部记录结构。relation类提供了一个通过存储引擎的handler类去读写磁盘数据的访问层。这个数组的元素现在有4个，前两个位置（0和1）分别对应着左、右两个子结点，后两个位置（2和3）是为排序和索引等临时性的关系操作预留的。

注解 relations数组的元素个数现在是4，这意味着你在DBXP SQL命令里最多可以使用4个表。如果你要对涉及4个以上的表的查询进行处理，需要修改稍后给出的内部查询表示转换代码，让它接受4个以上的表。

fields属性是一个指向MySQL中的Item类的指针，Item类包含着表的字段。它在投影操作和为关系操作管理各种必要的属性（比如说，去除那些满足表达式但与结果集无关的属性）时很有用。

需要说明的最后一个变量是preempt_pipeline变量，DBXP中的Execute类将使用它来实现一个循环以处理来自子结点的数据。凡是需要对一组数据（行）进行遍历处理的场合都离不开循环。比如说，如果一个根据公共属性联结两个表的联结操作不使用索引，但允许重新排序，它将需要遍历一个或两个子结点，才能完成正确的联结动作。

这个类还负责进行查询优化（见第11章）。因为查询树提供了处理查询树所需要的所有操作，而查询优化也是一组树操作，所以我决定把查询树结构和优化器方法封装在同一个类（叫作查询树类）里，然后调用那些方法去完成优化工作。

优化器方法实现了一种启发式算法（见第11章）。这些方法的执行将导致结点在查询树里的位置发生变化，或是把一个结点分成两个或更多个结点，它们的共同目标是生成一棵更优化的树。经过优化的查询树可以让查询更有效率地执行。

这个类还支持基于开销的优化技术，它使用了一种树遍历算法，为每个叶结点提供开销最小的索引和数据访问方法（因为对关系数据库的直接访问都发生在叶结点上）。

这个结构支持许多种关系操作，其中包括限制、投影、联结、集合和排序。查询结点结构可以把这些操作中的每一个表示为查询树里的一个结点，完成这些操作所需要的数据和信息也存储在各有关结点里。DBXP项目里的EXPLAIN命令采用了一种对查询树进行后序遍历的算法，从叶结点开始输出每一个结点的内容。MySQL里的EXPLAIN命令是用一组复杂的方法实现出来的，需要更多的运算时间才能执行完毕。

综上所述，查询树不仅是一种可以用来表示任何查询的内部表示，还提供了一种通过对树进行处理来进行查询优化的机制。事实上，树结构本身很容易优化，再通过把查询操作表示为树结构里的一个结点的办法，启发式优化器就更容易实现了。总之，这个查询树可以用在任何一种关系数据库系统里，并在经过进一步完善后可用在正式的软件产品里。

10.2 在MySQL里实现DBXP查询树

本节将演示如何把DBXP查询树结构添加到MySQL源代码里。创建一种关系数据库研究工具的第一步说明了查询树的工作原理，以及如何把MySQL查询结构转换为一棵基本的查询树（未经优化）。第10章和第11章将描述优化器和执行引擎。接下来的几节将演示如何添加查询树，以及如何为执行

SELECT和EXPLAIN SELECT命令添加一些存根。

10.2.1 被添加和修改的文件

本章的例子需要创建一些文件和修改一些MySQL源代码文件。表10-2列出了需要创建和修改的文件。

表10-2 需要创建和修改的文件

文 件	说 明
mysqld.cc	把DBXP版本号追加到MySQL版本号的后面
lex.h	把DBXP记号添加到词法散列表里
query_tree.h	DBXP查询树的头文件（新文件）
query_tree.cc	DBXP查询树的类文件（新文件）
sql_dbxp_parse.cc	DBXP解析器的辅助代码文件（新文件）
sql_lex.h	把DBXP SQL命令添加到符号表里
sql_yacc.yy	把DBXP SQL命令分析代码添加到解析器
sql_parse.cc	把处理新命令的代码添加到sql_parse.cc文件中的超大型switch语句里

10.2.2 创建测试

本节解释了为SELECT DBXP命令、查询树类和EXPLAIN SELECT DBXP以及SELECT DBXP生成存根的全过程。其目的是为了让用户输入任意合法的SELECT命令，处理这些查询和看到查询结果。

注解 因为DBXP引擎只是一个实验用途的引擎，所以它只能对基本的数据检索操作进行处理。为了把本章和后面几章的篇幅和复杂性控制在一个合理的范围内，我没有为DBXP引擎增加对HAVING、GROUP BY和ORDER BY等子句进行处理的功能。（有兴趣的读者可以自行实现这些操作。）

接下来的几个小节将详细介绍为实现这三个目标而添加DBXP代码的步骤。与其创建三个小测试，我决定只创建一个测试文件并用它来测试这三个方面。对于那些还没有实现的操作，可以把相应的查询语句改成注释（在语句开头加上一个#字符）。或者，也可以把它们留在那里进行测试并忽略那些因为尚未实现的命令而产生的出错消息。代码清单10-2给出了ExpertMySQLCh10.test文件的源代码。

代码清单10-2 本章使用的测试文件（ExpertMySQLCh10.test）

```
#
# Sample test to test the SELECT DBXP and EXPLAIN SELECT DBXP commands
#

# Test 1: Test stubbed SELECT DBXP command.
SELECT DBXP * FROM no_such_table;

# Test 2: Test stubbed Query Tree implementation.
SELECT DBXP * FROM customer;
```

```
# Test 3: Test stubbed EXPLAIN SELECT DBXP command.
EXPLAIN SELECT DBXP * FROM customer;
```

当然，可以在这个测试文件的基础上添加一些命令来研究那些新代码。通过MySQL Test Suite工具运行这类测试的详细步骤见第4章。

10.2.3 为 SELECT DBXP 命令生成存根

这一节将讲述如何把一条定制的SELECT命令添加到MySQL解析器里。你将看到如何修改MySQL解析器才能让它识别并接受一条模仿MySQL里传统的SELECT命令的新命令。

1. 标识修改

首先，应该给使用了DBXP技术的MySQL服务器加上一个识别标签。我选择的办法是给MySQL的版本号追加一个字符串，它可以让读者一眼看出正在连接的是我们修改过的服务器。

提示 你可以随时使用SELECT VERSION()命令来查看服务器的版本号。如果你正在使用MySQL命令行客户端程序，还可以修改它的命令提示符，让它把服务器的版本号显示在命令提示符里。这样一来，你随时都可以知道是不是在使用着添加了DBXP代码的服务器了。

给MySQL的版本号追加一个字符串的具体做法是：打开mysqld.cc文件并找到set_server_version()方法，添加一条语句把你的字符串追加到MySQL版本号字符串的后面。代码清单10-3给出了完成这个修改后的set_server_version()方法。

代码清单10-3 修改mysqld.cc文件

```
static void set_server_version(void)
{
    char *end= strxmov(server_version, MYSQL_SERVER_VERSION,
                      MYSQL_SERVER_SUFFIX_STR, Nulls);
#ifdef EMBEDDED_LIBRARY
    end= strmov(end, "-embedded");
#endif
#ifdef DBUG_OFF
    if (!strstr(MYSQL_SERVER_SUFFIX_STR, "-debug"))
        end= strmov(end, "-debug");
#endif
    if (opt_log || opt_update_log || opt_slow_log || opt_bin_log)
        strmov(end, "-log"); // This may slow down system
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the DBXP version number to the MySQL version number. */
    strmov(end, "-DBXP-1.0");
/* END DBXP MODIFICATION */
}
```

2. 修改词法结构

现在，我们来添加识别SELECT DBXP命令所需要的记号。打开lex.h文件并把代码清单10-4里的黑体字代码添加到symbols数组里去。

代码清单10-4 修改lex.h文件

```
static SYMBOL symbols[] = {  
/* BEGIN DBXP MODIFICATION */  
/* Reason for Modification: */  
/* This section identifies the symbols and values for the DBXP tokens */  
  { "WITH_DBXP_QUERYTREE", SYM(DBXP_SYM)},  
  { "DBXP", SYM(DBXP_SYM)},  
/* END DBXP MODIFICATION */
```

现在该生成词法散列表了。阅读后面与你的操作系统相关的那一节。继续这个过程，直到你得到一个可正常使用的gen_lex_hash命令行工具。

3. 在Windows平台上生成新的词法散列表

首先，打开主解决方案并把/sql子目录里的gen_lex_hash项目添加到其中。然后，把这些库添加到项目依赖关系里：dbug、libmysql、mysys、strings、taocrypt、yassl和zlib。接下来编译这个项目。编译器将把没有处理过的库文件包括进去。编译完成后，打开一个命令窗口，进入MySQL源代码根目录下的/sql子目录，然后运行gen_lex_hash.exe程序生成词法散列表，如下所示。

```
gen_lex_hash > lex_hash.h
```

这将生成一个新的lex_hash.h文件，用它编译出来的MySQL服务器才能识别刚才添加到lex.h文件里的符号。

4. 在Linux平台上生成词法散列表

Linux用户应该感到高兴，因为MySQL服务器的编译脚本可以替你完成编译gen_lex_hash工具的工作。这个工具放在一个名为gen_lex_hash子目录里并有它自己的制作文件。运行gen_lex_hash工具就可以生成词法散列表，如下所示。

```
gen_lex_hash > lex_hash.h
```

这将生成一个新的lex_hash.h文件，用它编译出来的MySQL服务器才能识别出你刚才添加到lex.h文件里的符号。不过，其实用不着进行这个步骤，因为适用于Linux平台的MySQL源代码发行版本里的制作文件会替你完成这个步骤。如果你想运行这个命令，唯一的理由可能是想看看编译过程有没有错误吧。

5. 添加SQL命令

本节解释了如何把新的SELECT DBXP命令添加到MySQL解析器里去。首先，你需要在sql_parse.cc文件中的解析器命令开关里为新命令增加一个新的case语句。那个开关使用枚举值作为case标号。为了添加一个新的case分支，你必须添加一个新的枚举值。解析器在识别出这些枚举值之后，将把它们存放到lex->sql_command成员变量里。给词法解析器添加一个新枚举值的具体做法是：打开sql_lex.h文件并把代码清单10-5里的黑体字代码添加到enum_sql_command枚举集合里。

代码清单10-5 为SELECT DBXP命令增加一个枚举值 (sql_lex.h文件)

```
enum enum_sql_command {  
...  
/* BEGIN DBXP MODIFICATION */  
/* Reason for Modification: */
```

```
/* This section captures the enumerations for the DBXP command tokens */
SQLCOM_DBXP_SELECT,
/* END DBXP MODIFICATION */
```

6. 把SELECT DBXP命令添加到MySQL解析器

为case语句添加了新的枚举值以后，还需要在解析器代码（sql_yacc.yy文件）里添加一些代码来识别新的SELECT DBXP语句。需要在解析器里增加一个新的记号，这样才能让解析器把MySQL版SELECT语句和DBXP版SELECT语句区分开。办法之一是通过编程让解析器在识别出新记号之后把sql_command变量设置为SQLCOM_DBXP_SELECT，而不是对应于MySQL版SELECT命令的枚举值SQLCOM_SELECT。这个技巧可以让你向正常的MySQL代码和新增加的DBXP代码发出含义相同的SELECT语句。举个例子，下面两条SELECT语句将完成同样的任务；它们的区别只是被优化得不一样而已。第一条SELECT命令将由标号为SQLCOM_SELECT的case分支负责处理，第二条将由标号为SQLCOM_DBXP_SELECT的case分支负责处理。

```
SELECT * FROM customer;
SELECT DBXP * FROM customer;
```

用来添加新记号的代码如代码清单10-6所示。在sql_yacc.yy文件里找到记号列表并把这些代码添加进去（这个记号列表里的元素基本上是按照字母表顺序排列的）。

代码清单10-6 把命令符号添加到解析器（sql_yacc.yy文件）

```
%token DAY_SYM
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section defines the tokens for the DBXP commands */
%token DBXP_SYM
/* END DBXP MODIFICATION */
%token DEALLOCATE_SYM
```

代码清单10-7给出了用来识别和处理SELECT DBXP命令的解析器代码。请注意，解析器需要识别SELECT和DBXP两个符号，然后需要对查询选项、字段表、FROM子句进行处理。接下来，解析器设置了sql_command变量。还请注意这段代码在原来的SELECT命令解析器代码前加了一个垂线字符|。这个字符是解析器语法用来处理命令变体的“或”操作符的。对解析器进行变体改动的具体做法是：打开sql_yacc.yy文件并找到select:标号，然后把代码清单10-7里的代码添加进去。

代码清单10-7 把命令语法操作添加到解析器（sql_yacc.yy文件）

```
select:
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section captures (parses) the SELECT DBXP statement */
SELECT_SYM DBXP_SYM DBXP_select_options DBXP_select_item_list
DBXP_select_from
{
    LEX *lex= Lex;
    lex->sql_command = SQLCOM_DBXP_SELECT;
}
```

```

/* END DBXP MODIFICATION */
| select_init
{
    LEX *lex= Lex;
    lex->sql_command= SQLCOM_SELECT;
}
;

```

请注意，代码清单10-7里的代码引用了几个其他的标号。代码清单10-8包含了这些操作的代码。第一个标号是DBXP_select_options，它负责识别SELECT命令的合法选项。虽然与MySQL的查询选项很相似，但它只支持两个选项：DISTINCT和COUNT(*)。下一个操作是DBXP_select_from代码，它负责识别FROM子句里的表，还调用了DBXP_where_clause操作来识别WHERE子句。下一个操作是DBXP_select_item_list，它与MySQL代码没有不同。最后，DBXP_where_clause操作负责识别WHERE子句里的参数。请仔细阅读下面这段代码并沿着它们的标号去看看都是干什么用的。把这段代码添加到解析器的具体做法是：找到select_from标号，然后把这段代码添加到它的上面——把这段代码添加到什么地方其实并不重要，但这个位置最合理，因为MySQL中的选择操作也在这个区域。代码清单10-8给出了SELECT DBXP解析器代码的全部源代码。

代码清单10-8 为SELECT DBXP命令添加额外的操作 (sql_yacc.yy文件)

```

/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section captures (parses) the sub parts of the SELECT DBXP statement */

DBXP_select_options:
/* empty */
|
/* Allow the distinct command switch */
DISTINCT { Select->options|= SELECT_DISTINCT; }
|
/* Enable the count(*) operation */
COUNT_SYM '(' '*' ') '
{
    /* Here we want to add the "count(*)" as an item field */
    THD *thd= YYTHD;
    if (add_item_to_list(thd,
        new Item_field(&thd->lex->current_select->context,
            NULL,NULL,"COUNT(*)"))
        YYABORT;
    }
;
/* The following sections define the rest of the SELECT command tokens */
DBXP_select_from:
    FROM join_table_list DBXP_where_clause;
/* parse the items in the select list (fields) */
DBXP_select_item_list:
/* empty */

```



```

| DBXP_select_item_list ',' select_item
| select_item
| '*'
{
    THD *thd= YYTHD;
    if (add_item_to_list(thd,
        new Item_field(&thd->lex->current_select->context,NULL,NULL,"*"))
        YYABORT;
};
/* process the where clause capturing the expressions */
DBXP_where_clause:
/* empty */ { Select->where= 0; }
| WHERE expr
{
    Select->where= $2;
    if ($2)
        $2->top_level_item();
}
;

/* END DBXP MODIFICATION */

select_from:
...

```

把词法解析器改好以后，还需要把它编译成C源代码。你可以用Bison工具来生成这些文件。打开一个命令窗口，进入MySQL源代码树根目录下的/sql子目录，然后执行下面这条命令。

```
bison -y -d sql_yacc.yy
```

这将生成两个新文件：y.tab.c和y.tab.h。这些文件将分别替代sql_yacc.cc和sql_yacc.h文件。在复制它们之前，应该先为原来的文件制作一个备份。在备份好文件之后，把y.tab.c复制为sql_yacc.cc（Windows用户需要复制为sql_yacc.cpp），把y.tab.h复制为sql_yacc.h。

注解 使用Linux平台的程序员不需要进行这个步骤，因为适用于Linux平台的MySQL源代码发行版本里的制作文件会替你们完成这一步。如果你想运行这个命令，唯一的理由大概是想看看编译过程中没有错误吧。比较新的MySQL源代码发行版本（如5.1）也包括了这个步骤。

Lex、YACC和Bison

Lex的含义是lexical analyzer generator（词法解析器生成器），它是一个用来把语句/表达式分解成一系列最基本的记号和常数的解析器，它也可以完成一些语法检查工作。YACC的含义是yet another compiler compiler（一个编译器的编译器），它可以根据你用YACC代码写出来的语法定义为你“新”语言生成一个编译器。这两个工具再加上Bison（一个能够从Lex/YACC代码生成C源代

码的YACC编译器),可以帮助人们简便快速地开发出一个能够分析和处理语言命令的子系统。MySQL的解析器就是一个经典的例子。

Windows程序员在编译sql_yacc.cpp(.cc)文件时,可能会遇到“某某符号未定义”的错误。如果你遇到的错误是yyerror或MySQLparse未定义,请打开sql_yacc.cpp(.cc)文件把代码清单10-9里的语句添加进去,然后重新运行Bison命令生成y.tab.c和y.tab.h文件。

代码清单10-9 缺少的#define语句

```
/* If NAME_PREFIX is specified substitute the variables and function
   names. */
#define yyparse MySQLparse
#define yylex MySQLlex
#define yyerror MySQLerror
#define yylval MySQLlval
#define yychar MySQLchar
#define yydebug MySQLdebug
#define yynerrs MySQLnerrs
```

7. 为SELECT DBXP命令生成存根

如果现在就去编译服务器,则在执行SELECT DBXP命令时不会遇到错误,但也仅此而已,其他什么事情也不会发生。这是因为你还需要在sql_parse.cc文件中的解析器开关里添加一个case分支。因为还没有编译DBXP引擎,所以现在只能生成一个case语句存根。代码清单10-10给出了可以用来实现SELECT DBXP命令的全套脚手架代码。在这段代码里,我利用了几个MySQL工具类创建了一个结果集。这段代码的第一部分为假想的表建立了一份字段清单。接下来的代码将把一些数据值写到网络流并在最后向客户端程序发送一个“文件尾”(end-of-file)标记。把数据写到输出流需要调用protocol->prepare_for_resend()方法,用protocol->store()方法把将要发送的数据存入缓冲区,最后用protocol->write()方法把缓冲区写到输出流。

代码清单10-10 修改解析器命令开关(sql_parse.cc文件)

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SELECT DBXP command. */
case SQLCOM_DBXP_SELECT:
{
    List<Item> field_list;
    /* The protocol class is used to write data to the client. */
    Protocol *protocol= thd->protocol;

    /* Build the field list and send the fields to the client */
    field_list.push_back(new Item_int("Id", (longlong) 1, 21));
    field_list.push_back(new Item_empty_string("LastName", 40));
    field_list.push_back(new Item_empty_string("FirstName", 20));
    field_list.push_back(new Item_empty_string("Gender", 2));
    if (protocol->send_fields(&field_list,
                            Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DEBUG_RETURN(TRUE);
```

```

protocol->prepare_for_resend();

/* Write some sample data to the buffer and send it with write() */
protocol->store((longlong)3);
protocol->store("Flintstone", system_charset_info);
protocol->store("Fred", system_charset_info);
protocol->store("M", system_charset_info);
if (protocol->write())
    DBUG_RETURN(TRUE);

protocol->prepare_for_resend();
protocol->store((longlong)5);
protocol->store("Rubble", system_charset_info);
protocol->store("Barnie", system_charset_info);
protocol->store("M", system_charset_info);
if (protocol->write())
    DBUG_RETURN(TRUE);

protocol->prepare_for_resend();
protocol->store((longlong)7);
protocol->store("Flintstone", system_charset_info);
protocol->store("Wilma", system_charset_info);
protocol->store("F", system_charset_info);
if (protocol->write())
    DBUG_RETURN(TRUE);

/*
    send_eof() tells the communication mechanism that we're finished
    sending data (end of file).
*/
send_eof(thd);
break;
}
/* END DBXP MODIFICATION */
case SQLCOM_PREPARE:
...

```

现在，当检测到SELECT DBXP命令时，这段存根代码将向客户端程序返回一个模拟的记录集。把这些代码添加好以后，就可以编译MySQL服务器并运行测试了。

8. 测试SELECT DBXP命令

这次运行的测试是发出一条SELECT DBXP命令，然后证实该语句由新的case语句存根进行解析和处理。你可以运行早创建的测试，也可以在MySQL命令行客户端工具里输入一条如下所示的命令（千万别忘了输入DBXP部分）。

```
SELECT DBXP * from no_such_table;
```

在DBXP后面输入什么并不重要，只要它是一条合法的SQL SELECT语句就行。代码清单10-11给出了一个符合预期的输出示例。

代码清单10-11 存根测试的结果

```
mysql> select DBXP * from no_such_table;
```

```
+-----+-----+-----+-----+
| Id | LastName | FirstName | Gender |
+-----+-----+-----+-----+
| 3 | Flintstone | Fred      | M      |
| 5 | Rubble    | Barnie    | M      |
| 7 | Flintstone | Wilma     | F      |
+-----+-----+-----+-----+
3 rows in set (0.23 sec)
```

```
mysql>
```

10.2.4 添加查询树类

既然已经有了SELECT DBXP命令存根，就可以开始实现DBXP特定的代码去执行SELECT命令了。本节将演示如何添加基本的查询树类（Query_tree类），以及如何把MySQL内部结构转换为查询树，但查询树的代码还要等到下一章才能最终完成。

1. 添加查询树头文件

添加查询树类需要创建查询树头文件并在MySQL代码里引用该头文件。这个查询树头文件的源代码如代码清单10-12所示。请注意，我把这个类命名为Query_tree；MySQL编程指南要求程序员使用首字母大写的单词来命名一个类。请大家先浏览一下这份代码清单。这些代码不算很长——只定义了一个基本的查询树结点结构和几个枚举集合。那些枚举集合定义了这几样东西：结点类型、联结条件类型、联结类型和集合类型。这些枚举集合可以让查询树上的各个结点在查询的执行过程中发挥独一无二的作用。下一章将对这些枚举集合的用途和用法做详细的解释。

你可以随意选择一种办法来创建（或下载）这个文件。请把它命名为query_tree.h，并把它放到MySQL源代码树中的/sql子目录里。现在还不用考虑怎样把它添加到DBXP项目里去，过一会儿会告诉你该怎么做。

代码清单10-12 查询树头文件

```
/*
Query_tree.h

DESCRIPTION
This file contains the Query_tree class. It is responsible for containing the
internal representation of the query to be executed. It provides methods for
optimizing and forming and inspecting the query tree. This class is the very
heart of the DBXP query capability! It also provides the ability to store
a binary "compiled" form of the query.

NOTES
The data structure is a binary tree that can have 0, 1, or 2 children. Only
Join operations can have 2 children. All other operations have 0 or 1
```

children. Each node in the tree is an operation and the links to children are the pipeline.

SEE ALSO

query_tree.cc

*/

#include "mysql_priv.h"

class Query_tree

{

public:

/*

This enumeration lists the available query node (operations)

*/

enum query_node_type

{

qntUndefined = 0,

qntRestrict = 1,

qntProject = 2,

qntJoin = 3,

qntSort = 4,

qntDistinct = 5

};

/*

This enumeration lists the available join operations

*/

enum join_con_type

{

jcUN = 0,

jcNA = 1,

jcON = 2,

jcUS = 3

};

/*

This enumeration lists the available join types

*/

enum type_join

{

jnUNKNOWN = 0, /* undefined */

jnINNER = 1,

jnLEFTOUTER = 2,

jnRIGHTOUTER = 3,

jnFULLOUTER = 4,

jnCROSSPRODUCT = 5,

jnUNION = 6,

jnINTERSECT = 7

};

enum AggregateType

/* used to add aggregate functions */


```

{
    atNONE      = 0,
    atCOUNT    = 1
};

/*
STRUCTURE query_node

DESCRIPTION
    This structure contains all of the data for a query node:

    NodeId -- the internal id number for a node
    ParentNodeId -- the internal id for the parent node (used for insert)
    SubQuery -- is this the start of a subquery?
    Child -- is this a Left or Right child of the parent?
    NodeType -- synonymous with operation type
    JoinType -- if a join, this is the join operation
    join_con_type -- if this is a join, this is the "on" condition
    Expressions -- the expressions from the "where" clause for this node
    Join Expressions -- the join expressions from the "join" clause(s)
    Relations[] -- the relations for this operation (at most 2)
    PreemptPipeline -- does the pipeline need to be halted for a sort?
    Fields -- the attributes for the result set of this operation
    Left -- a pointer to the left child node
    Right -- a pointer to the right child node
*/
struct query_node
{
    query_node();
    //query_node(const query_node &o);
    ~query_node();
    int          nodeid;
    int          parent_nodeid;
    bool         sub_query;
    bool         child;
    query_node_type node_type;
    type_join    join_type;
    join_con_type join_cond;
    COND        *where_expr;
    COND        *join_expr;
    TABLE_LIST *relations[4];
    bool        preempt_pipeline;
    List<Item>   *fields;
    query_node  *left;
    query_node  *right;
};

query_node *root;          //The ROOT node of the tree

~Query_tree(void);

```

```
void ShowPlan(query_node *QN, bool PrintOnRight);
};
```

光有查询树头文件还不够，还需要有一个查询树源代码文件。这个源代码文件起码需要包含查询树类的构造器和解构器的代码。代码清单10-13给出了完成后的构造器和解构器方法。创建一个名为query_tree.cc的文件并把这些代码输进去（或下载）。把这个文件放到MySQL源代码树中的/sql子目录里。我将在下一节告诉你怎样把它添加到DBXP项目里去。

注解 如果你使用的是Windows平台，应该把*.cc文件命名为*.cpp。

代码清单10-13 查询树类

```
/*
Query_tree.cc

DESCRIPTION
This file contains the Query_tree class. It is responsible for containing the
internal representation of the query to be executed. It provides methods for
optimizing and forming and inspecting the query tree. This class is the very
heart of the DBXP query capability! It also provides the ability to store
a binary "compiled" form of the query.

NOTES
The data structure is a binary tree that can have 0, 1, or 2 children. Only
Join operations can have 2 children. All other operations have 0 or 1
children. Each node in the tree is an operation and the links to children
are the pipeline.

SEE ALSO
query_tree.h
*/
#include "query_tree.h"

Query_tree::query_node::query_node()
{
    where_expr = NULL;
    join_expr = NULL;
    child = false;
    join_cond = Query_tree::jcUN;
    join_type = Query_tree::jnUNKNOWN;
    left = NULL;
    right = NULL;
    nodeid = -1;
    node_type = Query_tree::qntUndefined;
    sub_query = false;
    parent_nodeid = -1;
}

Query_tree::query_node::~~query_node()
```

```

{
    if(left)
        delete left;
    if(right)
        delete right;
}
Query_tree::~Query_tree(void)
{
    if(root)
        delete root;
}

```

2. 从MySQL结构构建查询树

接下来,需要编写一些用来把MySQL内部查询结构转换为查询树的代码。我决定使用一个辅助文件来存放这些代码,不把它们添加到sql_parse.cc文件里。事实上,sql_parse.cc文件中的那个超大型switch语句里的许多case分支所代表的SQL命令都是这么做的。请创建一个名为sql_dbxp_parse.cc的新文件,然后在这个文件里创建一个名为build_query_tree()的函数,这个函数的源代码如代码清单10-14所示。这些代码给出了一个基本的转换方法。请输入这些代码,你可以一边打字一边阅读它们(或把它们下载并粘贴到那个文件里)。

代码清单10-14 DBXP解析器辅助文件

```

/*
    sql_dbxp_parse.cc

    DESCRIPTION
        This file contains methods to execute the DBXP SELECT query statements.

    SEE ALSO
        query_tree.cc
*/
#include "query_tree.h"

/*
    Build Query Tree

    SYNOPSIS
        build_query_tree()
        THD *thd          IN the current thread
        LEX *lex          IN the pointer to the current parsed structure
        TABLE_LIST *tables IN the list of tables identified in the query

    DESCRIPTION
        This method returns a converted MySQL internal representation (IR) of a
        query as a query_tree.

    RETURN VALUE
        Success = Query_tree * -- the root of the new query tree.

```

```

    Failed = NULL
*/
Query_tree *build_query_tree(THD *thd, LEX *lex, TABLE_LIST *tables)
{
    DEBUG_ENTER("build_query_tree");
    Query_tree *qt = new Query_tree();
    Query_tree::query_node *qn =
        (Query_tree::query_node *)my_malloc(sizeof(Query_tree::query_node),
        MYF(MY_ZEROFILL | MY_WME));
    TABLE_LIST *table;
    int i = 0;
    int num_tables = 0;

    /* Create a new restrict node. */
    qn->parent_nodeid = -1;
    qn->child = false;
    /*
        Set the query type to unknown because we're creating a project node.
    */
    qn->join_type = (Query_tree::type_join) jnUNKNOWN;
    qn->nodeid = 0;
    qn->node_type = (Query_tree::query_node_type) qntProject;
    qn->left = 0;
    qn->right = 0;

    if(lex->select_lex.options & SELECT_DISTINCT)
    {
        //qt->set_distinct(true); /* placeholder for future modifications */
    }

    /* Get the tables (relations) */
    i = 0;
    for(table = tables; table; table = table->next_local)
    {
        num_tables++;
        qn->relations[i] = table;
        i++;
    }

    /* Populate attributes */
    qn->fields = &lex->select_lex.item_list;
    /* Process joins */
    if (num_tables > 0) //indicates more than 1 table processed
        for(table = tables; table; table = table->next_local)
            if ((table->on_expr != 0) && (qn->join_expr == 0))
                qn->join_expr = table->on_expr;
    qn->where_expr = lex->select_lex.where;
    qt->root = qn;
    DEBUG_RETURN(qt);
}

```


build_query_tree()函数将完成以下工作：创建一个新结点，识别查询将要到的表，填充字段清单，捕获join和where子句。这些东西都是执行最基本的查询命令所需要的。

3. 为查询树执行生成存根

接下来要添加的是用来创建查询树的代码。请创建一个名为DBXP_select_command()的函数并把代码清单10-15里的代码复制过去。把这个函数放到sql_DBXP_parse.cc文件里，刚才在sql_parse.cc文件里添加的case分支将调用这个方法。

代码清单10-15 处理SELECT DBXP命令

```
/*
  Perform Select Command

  SYNOPSIS
    DBXP_select_command()
    THD *thd                IN the current thread

  DESCRIPTION
    This method executes the command using the query tree and optimizer.

  RETURN VALUE
    Success = 0  /* Note: The use of 0 as success is a MySQL coding rule. */
    Failed = 1
*/
int DBXP_select_command(THD *thd)
{
  DEBUG_ENTER("dbxp_select_command");
  Query_tree *qt = build_query_tree(thd, thd->lex,
                                     (TABLE_LIST*) thd->lex->select_lex.table_list.first);
  List<Item> field_list;
  Protocol *protocol= thd->protocol;
  field_list.push_back(new
    Item_empty_string("Database Experiment Project (DBXP)",40));
  if (protocol->send_fields(&field_list,
                          Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
    DEBUG_RETURN(TRUE);
  protocol->prepare_for_resend();
  protocol->store("Query tree was built.", system_charset_info);
  if (protocol->write())
    DEBUG_RETURN(TRUE);
  send_eof(thd);
  delete qt;
  DEBUG_RETURN(0);
}
```

这段代码先调用了build_query_tree()函数，然后创建了一个结果集存根。这次创建的记录集只有一行和一系列，它的作用是向客户端传达这样一个信息：查询树已创建成功。虽说这些代码本身没什么趣味性，但它们可以让你对查询树做更多的实验（我在本章的最后给出了一些练习题）。最后，请把你创建的sql_DBXP_parse.cc文件放到MySQL源代码树的/sql子目录里。

4. 回顾为SELECT DBXP命令生成存根

打开sql_parse.cc文件并为DBXP_select_command()函数添加一个函数声明，请把这个函数声明放到标识符mysql_select_command的附近。代码清单10-16给出了DBXP_select_command()函数完整的函数声明，请把这些代码输入到如下所示的注释块上面。

代码清单10-16 修改解析器命令代码

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SELECT DBXP command. */
int DBXP_select_command(THD *thd);
/* END DBXP MODIFICATION */

/*****
** mysql_execute_command
** Execute command saved in thd and current_lex->sql_command
*****/
```

现在可以把那个case分支（也叫作解析器命令开关）里的代码改成调用这个新的DBXP_select_command()函数了。代码清单10-17给出了调用这个函数所需要的代码。请注意，只需要传递一个参数，那就是当前线程（thd）。MySQL的内部查询结构和其他必要的元数据都是通过这个线程指针引用的。正如你看到的那样，这个技巧让case语句变得非常简明。它还有助于DBXP代码的模块化，让那些代码更加容易维护和修改。

代码清单10-17 修改解析器命令开关（sql_parse.cc）

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SELECT DBXP command. */
case SQLCOM_DBXP_SELECT:
{
    res = DBXP_select_command(thd);
    if (res)
        goto error;
    break;
}
/* END DBXP MODIFICATION */
case SQLCOM_PREPARE:
{
    ...
}
```

在编译MySQL服务器之前，还需要把这些新创建的源代码文件（query_tree.h、query_tree.cc和sql_DBXP_parse.cc）添加到项目文件（Windows）或制作文件（Linux）里去。

5. 把文件添加到制作文件里（Linux）

在Linux平台上添加项目文件，需要修改MySQL源代码树根目录下的/sql子目录里的Makefile.am文件。打开这个文件并找mysqld_SOURCES标签。把刚才提到的那几个新源代码文件添加到编译服务器项目（mysqld）的源代码文件清单里去。代码清单10-18给出了这份清单的开头部分，最后一行是刚添加的新项目文件。

代码清单10-18 修改Makefile.am文件

```

mysqld_SOURCES = sql_lex.cc sql_handler.cc sql_partition.cc \
    item.cc item_sum.cc item_buff.cc item_func.cc \
    item_cmpfunc.cc item_strfunc.cc item_timefunc.cc \
    thr_malloc.cc item_create.cc item_subselect.cc \
    item_row.cc item_geofunc.cc item_xmlfunc.cc \
    field.cc strfunc.cc key.cc sql_class.cc sql_list.cc \
    net_serv.cc protocol.cc sql_state.c \
    lock.cc my_lock.c \
    sql_string.cc sql_manager.cc sql_map.cc \
    mysqld.cc password.c hash_filo.cc hostname.cc \
    set_var.cc sql_parse.cc sql_yacc.yy \
    sql_dbxp_parse.cc query_tree.cc \
...

```

注意 在修改制作文件时,如果你想让文件里的清单整齐美观,一定要使用空格来排版,不要使用制表符。

6. 把文件添加到mysqld项目里 (Windows)

在Windows平台上添加项目文件是一件很简单的事。在Visual Studio里用鼠标右击mysqld项目,然后执行菜单命令Add►Existing Item,把新文件(query_tree.h、query_tree.cc和sql_DBXP_parse.cc)添加进去就行了。当编译mysqld项目的时候,这些新源代码文件会随着其他文件一起被编译。

7. 测试查询树

在编译好服务器之后,就可以用一条SQL命令来测试它了。和刚才那次测试不同,这次需要输入一条合法的SQL命令来引用存在的对象。你可以像前面那样运行这个测试(如代码清单10-19所示),也可以在MySQL命令行客户端程序里输入下面这条命令。

```
SELECT DBXP * from customer;
```

代码清单10-19 SELECT DBXP测试的结果

```
mysql> SELECT DBXP * FROM customer;
```

```

+-----+
| Database Experiment Project (DBXP) |
+-----+
| Query tree was built.               |
+-----+
1 row in set (0.00 sec)

```

```
mysql>
```

我们已经生成了SELECT DBXP操作存根并构建了查询树,但事情还没有结束。要是想办法看到查询的真面目就好了。让我们来创建一个函数去完成EXPLAIN命令的工作,但不把信息列成表格,我们将把查询以图形的方式^①显示成一棵树的样子。

① 不管怎样,在命令行接口的能力范围内。

10.2.5 显示查询树的细节

添加一个新命令需要在sql_parse.cc文件中的解析器命令开关里添加一个新的case语句，为这个新case语句添加一个新的枚举值并添加一些解析器代码来识别新命令。还需要在sql_DBXP_parse.cc文件里添加一些代码来执行新命令。为解析器创建并增加一个EXPLAIN命令来解释查询树，听起来可能有些复杂，但MySQL里有一个现成的EXPLAIN SELECT命令可以借鉴，我们只要把它的代码复制过来，再做些修改就行了。

1. 把EXPLAIN SELECT DBXP命令添加到MySQL解析器

首先，要给词法解析器添加一个新枚举值。打开sql_lex.h文件，把一个名为SQLCOM_DBXP_EXPLAIN_SELECT的枚举值添加到SQLCOM_DBXP_SELECT枚举值的后面。代码清单10-20给出了完成这一修改后的代码片段。添加好新的枚举值后，就可以按照前面介绍的步骤去生成新的词法散列表了。

代码清单10-20 添加EXPLAIN枚举值 (sql_lex.h文件)

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section captures the enumerations for the DBXP command tokens */
SQLCOM_DBXP_SELECT,
SQLCOM_DBXP_EXPLAIN_SELECT,
/* END DBXP MODIFICATION */
```

接下来是向解析器添加一些代码。打开sql_yacc.yy文件并找到describe:标签。代码清单10-21给出了为新EXPLAIN命令添加的解析器代码。把这些代码输入到describe:标签后、原来那些MySQL代码之前。不要忘记在原来的EXPLAIN代码的前面加上一个垂线字符|。

代码清单10-21 为EXPLAIN命令修改解析器代码

```
describe:
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section captures (parses) the EXPLAIN (DESCRIBE) DBXP statements */
describe_command SELECT_SYM DBXP_SYM DBXP_select_options
  DBXP_select_item_list DBXP_select_from
{
  LEX *lex= Lex;
  lex->lock_option= TL_READ;
  lex->sql_command = SQLCOM_DBXP_EXPLAIN_SELECT;
  lex->select_lex.db= 0;
  lex->verbose= 0;
}

/* END DBXP MODIFICATION */

| describe_command table_ident
...
```

这些代码将使解析器可以识别出EXPLAIN SELECT DBXP命令。正如你看到的那样，它调用了许多与SELECT DBXP命令的解析器代码相同的操作，唯一的区别是这些代码将把sql_command变量设置为刚刚

添加的新枚举值 (SQLCOM_DBXP_EXPLAIN_SELECT)。另一个需要注意的地方是，添加在原来那些代码前面的垂线字符|，它在解析器语法里被解释为“或”操作符。

修改sql_parse.cc文件里的解析器开关语句，需要为sql_DBXP_parse.cc文件里负责执行EXPLAIN命令的代码添加一个函数声明。打开sql_parse.cc文件并为EXPLAIN函数添加一个函数声明。把这个函数命名为DBXP_explain_select_command() (看出规律了吗?)，再将其添加到DBXP_select_command()函数声明的后面。代码清单10-22是为两条DBXP新命令添加函数声明后的完整代码。

代码清单10-22 修改解析器命令代码

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SELECT DBXP command. */
int DBXP_select_command(THD *thd);
int DBXP_explain_select_command(THD *thd);
/* END DBXP MODIFICATION */

/*****
** mysql_execute_command
** Execute command saved in thd and current_lex->sql_command
*****/
```

还需要为DBXP版EXPLAIN命令添加新的case分支。这个新分支与我们在前面为SELECT DBXP命令添加的case分支大同小异。代码清单10-23是添加了这个新case分支后的代码片段。

代码清单10-23 修改解析器开关语句

```
/* BEGIN DBXP MODIFICATION */
/* Reason for Modification: */
/* This section adds the code to call the new SELECT DBXP command. */
case SQLCOM_DBXP_SELECT:
{
    res = DBXP_select_command(thd);
    if (res)
        goto error;
    break;
}
case SQLCOM_DBXP_EXPLAIN_SELECT:
{
    res = DBXP_explain_select_command(thd);
    if (res)
        goto error;
    break;
}
/* END DBXP MODIFICATION */
```

2. 创建show_plan()函数

EXPLAIN SELECT DBXP命令将把查询路径以字符图形的方式显示出来。有关的EXPLAIN代码是在sql_DBXP_parse.cc文件中的一个名为show_plan()的函数里执行的。为了让show_plan()函数的代码容

易阅读，我编写了一个名为write_printf()的辅助函数。代码清单10-24和代码清单10-25给出了这两个方法的完整代码。

代码清单10-24 添加一个函数来捕获协议存储和写入语句

```

/*
  Write to vio with printf.

  SYNOPSIS
    write_printf()
    Protocol *p      IN the Protocol class
    char *first      IN the first string to write
    char *last       IN the last string to write

  DESCRIPTION
    This method writes to the vio routines printing the strings passed.
  RETURN VALUE
    Success = 0
    Failed = 1
*/
int write_printf(Protocol *p, char *first, char *last)
{
    char *str = (char *)my_malloc(256, MYF(MY_ZEROFILL | MY_WME));

    DEBUG_ENTER("write_printf");
    strcpy(str, first);
    strcat(str, last);
    p->prepare_for_resend();
    p->store(str, system_charset_info);
    p->write();
    my_free((gptr)str, MYF(0));
    DEBUG_RETURN(0);
}

```

write_printf()函数调用protocol->store()和protocol->write()函数把一行字符图形输出到客户端，它的用途和用法可以从代码清单10-25给出的show_plan()函数的源代码里看出来。下一节为大家准备了一个执行这些代码的例子。show_plan()函数使用了一种后序遍历算法，从查询树的根结点开始生成一份查询计划。请把这两个方法添加到sql_DBXP_parse.cc文件里去。

代码清单10-25 show_plan()函数的源代码

```

/*
  Show Query Plan

  SYNOPSIS
    show_plan()
    Protocol *p      IN the MySQL protocol class
    query_node *Root IN the root node of the query tree
    query_node *qn   IN the starting node to be operated on.

```

bool print_on_right IN indicates the printing should tab to the right of the display.

DESCRIPTION

This method prints the execute plan to the client via the protocol class

WARNING

This is a RECURSIVE method!

Uses postorder traversal to draw the query plan

RETURN VALUE

Success = 0 /* The use of 0 as success is a MySQL coding rule */

Failed = 1

```

*/
int show_plan(Protocol *p, Query_tree::query_node *root,
              Query_tree::query_node *qn, bool print_on_right)
{
    DEBUG_ENTER("show_plan");

    /* spacer is used to fill white space in the output */
    char *spacer = (char *)my_malloc(80, MYF(MY_ZEROFILL | MY_WME));
    char *tblname = (char *)my_malloc(256, MYF(MY_ZEROFILL | MY_WME));
    int i = 0;

    if(qn != 0)
    {
        show_plan(p, root, qn->left, print_on_right);
        show_plan(p, root, qn->right, true);

        /* Draw incoming arrows */
        if(print_on_right)
            strcpy(spacer, "          |          ");
        else
            strcpy(spacer, "          ");

        /* Write out the name of the database and table */
        if((qn->left == NULL) && (qn->right == NULL))
        {
            /*
             * If this is a join, it has 2 children so we need to write
             * the children nodes feeding the join node. Spaces are used
             * to place the tables side-by-side.
             */
            if(qn->node_type == Query_tree::qntJoin)
            {
                strcpy(tblname, spacer);
                strcat(tblname, qn->relations[0]->db);
                strcat(tblname, ".");
                strcat(tblname, qn->relations[0]->table_name);
                if(strlen(tblname) < 15)

```

```

        strcat(tblname, "          ");
    else
        strcat(tblname, "          ");
        strcat(tblname, qn->relations[1]->db);
        strcat(tblname, ".");
        strcat(tblname, qn->relations[1]->table_name);
        write_printf(p, tblname, "");
        write_printf(p, spacer, "          |");
        write_printf(p, spacer, "          |-----");
        write_printf(p, spacer, "          |");
        write_printf(p, spacer, "          V  V");
    }
else
{
    strcpy(tblname, spacer);
    strcat(tblname, qn->relations[0]->db);
    strcat(tblname, ".");
    strcat(tblname, qn->relations[0]->table_name);
    write_printf(p, tblname, "");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          V");
}
}
else if((qn->left != 0) && (qn->right != 0))
{
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          |-----");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          V  V");
}
else if((qn->left != 0) && (qn->right == 0))
{
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          |");
    write_printf(p, spacer, "          V");
}
else if(qn->right != 0)
{
}
write_printf(p, spacer, "-----");

/* Write out the node type */
switch(qn->node_type)
{
case Query_tree::qntProject:
{
    write_printf(p, spacer, "          PROJECT          |");

```



```

        write_printf(p, spacer, "-----");
        break;
    }
    case Query_tree::qntRestrict:
    {
        write_printf(p, spacer, "|    RESTRICT    |");
        write_printf(p, spacer, "-----");
        break;
    }
    case Query_tree::qntJoin:
    {
        write_printf(p, spacer, "|    JOIN    |");
        write_printf(p, spacer, "-----");
        break;
    }
    case Query_tree::qntDistinct:
    {
        write_printf(p, spacer, "|    DISTINCT    |");
        write_printf(p, spacer, "-----");
        break;
    }
    default:
    {
        write_printf(p, spacer, "|    UNDEF    |");
        write_printf(p, spacer, "-----");
        break;
    }
}
write_printf(p, spacer, "| Access Method: |");
write_printf(p, spacer, "|    iterator    |");
write_printf(p, spacer, "-----");
if(qn == root)
{
    write_printf(p, spacer, "    |");
    write_printf(p, spacer, "    |");
    write_printf(p, spacer, "    V");
    write_printf(p, spacer, "    Result Set");
}
}
my_free((gptr)spacer, MYF(0));
my_free((gptr)tblname, MYF(0));
DEBUG_RETURN(0);
}

```

最后，还需要添加一些代码来执行EXPLAIN SELECT DBXP命令，调用show_plan()函数，并把结果返回给客户端。代码清单10-26给出了这个函数完整的源代码。请注意，这个函数先创建了一棵查询树，然后创建了一个只包含一个名为Excution Path的字符串类型列的字段清单，最后调用show_plan()函数把执行计划输出到客户端。

代码清单10-26 DBXP EXPLAIN命令的源代码

```

/*
  Perform EXPLAIN command.

  SYNOPSIS
    DBXP_explain_select_command()
    THD *thd                IN the current thread

  DESCRIPTION
    This method executes the EXPLAIN SELECT command.

  RETURN VALUE
    Success = 0
    Failed = 1
*/
int DBXP_explain_select_command(THD *thd)
{
  DEBUG_ENTER("dbxp_explain_select_command");
  Query_tree *qt = build_query_tree(thd, thd->lex,
                                     (TABLE_LIST*) thd->lex->select_lex.table_list.first);
  List<Item> field_list;
  Protocol *protocol= thd->protocol;
  field_list.push_back(new Item_empty_string("Execution Path",NAME_LEN));
  if (protocol->send_fields(&field_list,
                          Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
    DEBUG_RETURN(TRUE);
  protocol->prepare_for_resend();
  show_plan(protocol, qt->root, qt->root, false);
  send_eof(thd);
  delete qt;
  DEBUG_RETURN(0);
}

```

接下来编译MySQL服务器，并把它与测试文件放在一起运行。

3. 测试DBXP EXPLAIN命令

和以前的测试一样，你既可以使用MySQL Test Suite工具来运行这个测试，也可以在MySQL的命令行客户端里输入一条合法的SQL命令。代码清单10-27给出了查询执行计划的一份示例输出。因为查询命令现在还没有经过优化，所以你只能看到一个结点。等把优化器添加到DBXP项目里（见第11章）以后，查询执行计划才可以真实地反映出用户输入的查询语句将沿着怎样的路径得到执行。

代码清单10-27 对DBXP EXPLAIN进行测试的结果

```
mysql> EXPLAIN SELECT DBXP * FROM customer;
```

```

+-----+
| Execution Path |
+-----+
|      test.customer      |
+-----+

```

```

      |
      |
      | V
      |-----|
      | PROJECT |
      |-----|
      | Access Method: |
      |   iterator   |
      |-----|
      |
      | V
      | Result Set |
      |-----|
+-----+
15 rows in set (0.00 sec)

mysql>

```

这份输出报告比一份罗列事实的清单有意义多了。在这个阶段把EXPLAIN命令添加到DBXP项目中可以让你见证和诊断DBXP优化器是如何对查询树进行优化的。等你开始自己的实验时，就会发现它非常有帮助。

不过，在开始自己的实验之前，应该运行一次完整的测试，把你在本章时添加的三大部分代码全面地测试一遍。

10.3 小结

本章介绍了一些更复杂的数据库技术。文中讲解了查询命令在MySQL服务器内部是如何表示的，以及sql_parse.cc文件里的超大型switch语句是如何对它们进行分析和处理的。更重要的是本章阐述了如何利用MySQL和查询树类来设计自己的数据库内部组成实验。这些技术的知识可以让你对MySQL的内部组件如何构建有进一步的理解。

下一章将实现一种查询树优化策略并在这个过程中对查询的内部表示做更深入的讨论。你肯定很想知道关系数据库系统的优化器是如何构建的，在下一章就会看到一个利用本章完成的查询树类去创建一个启发式查询优化器的例子。

练习

下面是几个值得进一步研究的问题。它们在同类问题中很有代表性，在研究关系数据库技术时可以把它们作为实验题材（或课堂作业）。

1. 图10-1里的查询命令暴露出了某个表的设计方案有缺陷。你能发现这个缺陷吗？这个缺陷违反了某个范式吗？如果是，那么是哪一种范式？
2. 剖析TABLE结构并把SELECT DBXP存根改成可以返回关于表及其字段的信息。
3. 修改EXPLAIN SELECT DBXP命令，让它产生与MySQL的EXPLAIN SELECT命令类似的输出信息。
4. 修改build_query_tree()函数，让它可以识别并处理LIMIT子句。
5. 如何修改query_node结构才能让它支持HAVING、GROUP BY和ORDER BY子句？

第 10章完成的查询树类只是为DBXP项目创建一个实验性查询优化和执行引擎的第一步。本章将演示如何把优化器添加到这个查询树类里。本章先介绍在优化器里使用启发式规则的理由，然后开始编写代码。因为有些函数的代码相当冗长，所以本章中的代码示例不都是完整的源代码。如果想参照书中的例子做练习，建议你从本书的配套站点下载有关的源代码，不要从头开始输入这些代码。

11.1 查询优化器的类型

第一代优化器是为System R^①和INGRES^②等早期的数据库系统设计的。这些优化器是为关系模型的某种特定实现而开发的，它们经受住了时间的检验，并说明了如何去实现优化器。直到今天，还有许多商业数据库系统是建立在这些工作的基础上的。从那时起，随着对关系模型的扩展，已经为面向对象的数据库系统和分布式数据库系统创建了多种优化器。

Volcano优化器就是一个很典型的例子。它使用了一种动态编程算法在面向对象数据库系统中为基于开销的优化生成查询计划。另外一个例子与如何在超大型数据库系统（类似于分布式系统，但数据库和用户的分布情况相对比较集中）里进行查询优化有关，这类环境通常需要使用统计学方法来生成优化策略。

对查询优化技术提出独特要求的另一类场合是内存驻留型数据库系统。内存驻留型数据库系统指的是那些把整个系统和所有数据都保存在计算机的主存储器和辅助存储器（硬盘）里的数据库系统。这类系统大都属于嵌入式系统，但也有不少大型分布式系统是通过一组内存驻留型数据库系统来提供信息的。内存驻留型数据库系统对优化工作的要求是算法越快越好，花在查询优化环节的时间在整个查询处理时间里占的比例越小越好^③。

所有对传统和非传统的查询优化的研究工作都建立在System R的第一代优化器的基础上。System R的优化器是一种基于开销的优化器，它使用收集到的关于数据库和数据的信息或统计数据来估算各

-
- ① P. G. Selinger, M. M. Astraham, D. D. Chamberlin, R. A. Lories, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Aberdeen, Scotland: 23–34. 该书被一些人誉为“查询优化的圣经之作”。
 - ② M. Stonebraker, E. Wong, P. Kreps. 1976. The Design and Implementation of INGRES. *ACM Transactions on Database Systems* 1(3): 189–222.
 - ③ 传统系统中查询的执行，不仅包括查询的处理，还包括从物理介质访问数据。可是，常驻内存系统并没有与从物理介质获取数据相联系的较长的访问时间。

种执行计划的开销，并从中提出一个开销最小的加以执行。为同一个查询命令生成一系列不同但等效（它们生成的结果是一样的）的内部表示的概念就是从那时产生的，这个概念提供了一种用来保存那些可供选择的内部查询表示的机制。现在的人们把那些可供选择的内部查询表示称为查询计划。之所以选择开销最小的查询计划来执行，是因为它最有可能让查询命令的执行效率达到最高。

System R的主要特性之一是提出了“选择性”的概念——通过计算一个包含对属性及其值的引用的表达式来预测结果。选择性是确定合取选择中的简单表达式应该按什么顺序被测试的核心。选择性最小的表达式意味着检索出来的元组（行）个数是最少的。于是，这个表达式应当成为查询中的第一步操作的基础。合取选择可以看作关系代数里的“交”操作条件。反过来说，析取选择可以看作关系代数里的“并”操作条件。对于析取条件，各条件的先后顺序无关紧要。

有些查询优化器（比如System R）不处理所有可能的联结顺序，而只是搜索某些已知的可产生更高效的执行的联结顺序。比如，多路联结（multiway join）的执行顺序将被编排成最先执行那些生成最不可能的结果的条件。类似的，System R优化器按照“先执行其右操作数是其中一个初始关系（表或结果集）的联结操作”的原则来编排联结操作的顺序。人们把这种联结顺序称为“左优先”（left-deep）联结顺序。左优先联结顺序非常适合以流水线方式执行，因为右操作数通常都是一个已知的关系（而不是一个还没有被求值出来的中间结果），系统只需按顺序为每个联结操作提供一个输入源就可以了。流水线的使用是数据库实验项目所用到的优化器和执行引擎的重要因素之一。

11.1.1 基于开销的优化器

基于开销的优化器先根据各种等价规则为一个给定的查询生成一组查询评估计划，然后根据以前收集到的关于执行这个查询所需要的关系和操作的信息（或统计数据），从中选出一个开销最小的来执行。对于一个复杂的查询，有很多可行的计划与之对应。

基于开销的优化追求这样一个目标：尽可能多地利用索引和从以前的查询中收集来的统计信息来安排查询执行和表访问。Microsoft SQL Server和Oracle使用的都是基于开销的优化器。

人们把在数据库系统里负责收集和处理统计信息的部分（以及许多其他的工具函数）统称为数据库目录（database catalog）。这个目录存储和管理着关于每一个关系（表）和它的各种访问路径的统计信息。等到需要选择一条访问路径的时候，这些信息将被用来挑选一条效率最高（开销最小）的路径。比如说，System R会为每个表收集关于以下几个方面的信息。

- (1) 每个关系的基数（cardinality）。
- (2) 在存有每个关系的元组的内存段中页面的数目。
- (3) 在存有关系的元组（阻塞因子或填充符）的内存段中数据页的片段。
- (4) 对于每一个索引来说：
 - 每一个索引中不同的键的数目；
 - 每一个索引中页面的数目。

这些统计信息来自系统中的多个渠道，加载关系和创建索引时都会产生一些新的统计信息。这些信息可以定期使用一条用户命令来更新^①，任何用户都可以使用这条命令。System R不以实时方式更新这些信息，因为这样做可能在系统目录上产生额外的数据库操作和死锁瓶颈。动态更新这些统计信

^① 这种做法被大多数商业数据库系统一直延用到现在。

息需要访问一些表和修改一些内容，在多用户环境里这样做会降低系统处理并发查询的能力。

基于开销的优化器在如何使用收集来的统计信息方面并不复杂。我接触到的数据库专业人员往往会认为，收集和使用这些统计信息是一个非常复杂且对查询优化而言至关重要的过程。这也难怪，基于开销的查询优化和一些混合的优化方案，确实需要使用这些信息来评估各种查询计划，但这一过程既不复杂也算不上关键。只要想一想属性值呈均匀分布时的情况就会明白这个道理了。这个概念本身证明了统计应用的不准确性。统计计算本质上是一门从整体来分析数据分布情况的学问，无法产生一个精确的结果。统计信息只能帮助判断某个查询执行计划一般情况下是不是比另一个开销大。

属性值的频率分布是预测查询结果规模的常用手段。通过预测属性的可能（或实际^①）值的分布情况，数据库系统可以大致估算出某给定查询计划需要处理的元组（行）个数，进而估算出这个查询计划的总开销。不过，现代数据库系统往往只对单个属性的频率分布进行处理，因为分析属性的所有组合本身就是一件开销巨大的工作。这还与属性值独立的假设相对应，虽然这只在极少数情况下成立，但几乎所有的关系数据库系统都把它当作一个事实来接受。

收集分布数据需要不断更新那些统计信息，或对之进行预测分析。另一个办法是假设属性值总是呈均匀分布，每一种不重复的属性值都对应着所有不同的值。比如，假设某个表里有5000个元组，它的某个属性有50个不重复的值，那么每个不重复的属性值都被表示100次。这种情况在现实世界里是很少见的，而且常常不正确，但在没有任何统计信息可供参考的情况下，按照这一假设去进行估算也合情合理。

在最坏的情况下，动态编程所需要的内存和运行时间会随着查询规模（联结操作的个数）的增加而呈指数增加态势，因为每一步骤生成的所有可行的局部计划都必须保存起来供下一个优化步骤使用。许多现代数据库系统都对查询规模设置了一个上限（通常是15个联结操作左右），超过了这个限度的查询很容易因为内存消耗过大而导致优化器崩溃。从实际情况看，绝大多数查询的规模都不会超过10个联结操作，基于开销的优化器所使用的优化算法对这些查询的优化效果还是不错的。人们在评价一种查询优化搜索策略的好坏时往往以此为标准。下面是基于开销的优化器需要收集和使用的一些关于表（或关系）中的行（或元组）的统计信息：

- 表中元组的个数；
- 包含有行的块的个数（块计数）；
- 行的字节长度；
- 每种属性（或列）的不重复值的个数；
- 每种属性的选择基数（有时表示为均匀分布）；
- 索引的内部结点“扇出”（发展成子树的子结点的个数）；
- 索引的B树的高度；
- 索引的叶结点所容纳的块。

把最终结果写回到磁盘的开销通常被忽略。这是因为不管最后选用的是哪一个查询评估计划，这个开销都是固定不变的，是否把它计算在内对计划的选择没有影响。

现在大多数数据库系统都采用某种动态编程技术来生成所有可能的查询计划。这些动态编程技术为开销的优化器提供了好的性能，但这类算法往往相当复杂，在遇到比较复杂的查询时会消耗更多的

① 实时的统计信息累加称为背负式统计生成。

资源。虽说绝大多数数据库系统在日常使用中很少遇到这样的查询，但分布式数据库系统和高性能计算环境等领域的研究人员一直在寻找更好的动态编程技术。Kossmann和Stocker的最新研究成果表明，传统的查询优化技术已经开始显露出其局限性^①，更高效的优化技术可根据人们的实践而不是采用穷举策略来生成执行计划。换句话说，我们需要的优化器不仅要在各种随时都在变化的通用环境里表现出色，还要在一些独一无二的数据库环境里表现得同样出色。

11.1.2 启发式优化器

启发式优化的目标是应用一些规则来保证查询执行的效率。使用启发式优化器的系统包括INGRES和各种学术性变体。大部分系统都把启发式优化当作一种用来排除坏计划的手段，而不是把它用作主要的优化手段。

在选用某种查询计划之前，启发式优化器会使用一些关于如何先把查询变形为一种最优形式的规则。运用启发（即规则）的目的是为了把效率不高的查询计划排除掉。采用启发式为基础生成的查询计划确保其在求值之前最有可能（但并不绝对）得到优化。这类启发包括以下几种。

- 尽可能早执行选取操作，最好是在投影操作之前执行，因为这样可以减少需要发送到查询树的元组的个数。
- 尽可能早执行投影操作。
- 判断哪些选取操作和联结操作将生成最小的结果集并最先执行它们（最左优先策略）。
- 用联结操作代替笛卡儿积。
- 把投影操作所处理的属性尽量移动到树的最下端。
- 把那些可以采用流水线方式处理的子树识别出来。

启发式优化器并不是什么新技术，研究人员很早就为各种专用的系统创建出了基于规则的优化器，**Prairie**就是一个基于规则的优化器。这个基于规则的优化器允许使用一种给定的语言记号来创建规则，对查询的处理是在规则的控制下进行优化的。不过，**Prairie**优化器本质上仍是一个基于开销的优化器，它对规则的使用只是为了对优化器进行调优。

除了**Prairie**以及像INGRES这样的早期的基本组件之外，没有一种商业数据库系统实现了真正的启发式优化器。它们当中有些确实有一个基于规则（启发）的优化步骤，但那通常是作为传统的基于开销的优化器的一项额外功能或预处理功能实现的，或者是作为优化过程中的一个预处理步骤实现的。

11.1.3 语义优化器

语义优化的目标是生成利用数据库、关系和其中的索引的语义（或拓扑结构）来生成查询的查询执行计划，这保证了在给定的数据库里执行查询时可采用最佳实践方法。语义查询优化利用模式的知识（比如一致性约束）把查询命令转换为一种可以比原来更快地得到答案的新形式。

虽然还没有成为商业数据库系统中的主要优化技术，但语义优化目前已是热门的研究方向之一。语义优化操作需要优化器对实际的数据库模式有一个基本的了解。在收到用户提交的查询之后，优化器将根据它从系统各方面收集到的系统约束去简化它，或者忽略优化器判断出肯定会返回一个空结果

^① D. Kossmann, and K. Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Transactions on Database Systems* 25(1): 43–82.

集的特殊查询。人们对语义优化技术寄予厚望，也许在不久的将来，可以在某个关系数据库系统里看到这类技术在为进一步提高查询处理的效率而大显身手。

11.1.4 参数优化器

Ioannidis在他关于参数查询优化技术的论文里描述了一种把启发式优化策略与基于开销的优化策略相结合的查询优化方法。这种查询优化器的基本思路是：先生成一个较小的高效率查询计划的集合，然后为这个集合里的各查询计划估算其开销，最后选择一个开销最小的计划送去执行^①。查询计划是用一种名为sipR的随机算法生成的。使用了参数查询优化技术的系统可以在挑选查询计划的时候，把参数变化的不确定性（比如缓冲区的长度）也考虑进去，优化器将根据各有关参数来决定是即时生成一个最优计划，还是从以前保存的查询计划当中挑选一个。

Ioannidis在他的文章里提出了一个很有意思的观点：动态编程算法并不是必须的，因此使用这些技术而增加的开销是可以避免的。Ioannidis还认为，先运用一些规则对查询进行精简或变形，再使用动态编程算法来进行查询优化的数据库系统，都可以看作是System R系统的改进版本。Ioannidis提供的数据库表明，对于小查询（联结操作在10个左右或以下），动态编程算法要优于随机化算法；对于规模更大的查询，随机化算法要优于动态编程算法。

11.2 再次讨论启发式优化

启发式优化器通过一组预先定义好的规则来保证良好的执行计划。因此，启发式优化器选用的规则的好坏和完备性将直接决定它生成的查询计划的好坏。接下来的几个小节将对DBXP查询优化器所使用的优化规则进行描述。虽然这些规则非常基本，但它们对典型查询命令的优化效果接近最优，在确保查询结果的准确性的前提下有很高的执行速度。

首先，为查询树的初创工作添加几条基本的规则。简单地说，所有的执行都将发生在查询树的结点里。限制（restriction）和投影（projection）操作将在树枝上进行，不产生中间关系。联结（join）操作总是发生在两条路径的交汇点。多向联结（multiway join）被处理为一系列两路联结。为了生成高质量的查询计划，DBXP优化器将使用以下规则把查询树转换为一种能高效率执行的形式^②。

(1) 把包含着“投影和联结”或“选取和联结”操作的所有结点一分为二。这是因为有些查询命令会把联结条件写在WHERE子句里^③，而这将“欺骗”优化器为一个只有一部分是联结条件的表达式创建一个联结结点。

(2) 把所有的限制尽可能地向下移动到查询树的叶结点。根据各个关系把尽可能多的表达式归成一组，放到一个查询树结点里。有些复杂的表达式可能无法如此简化，但绝大多数表达式都可以简化为一个关系。把限制尽量下推到叶结点的目的是为了尽量减少需要在查询树中传递的元组个数。

(3) 把所有的投影操作尽可能地向下移动到查询树的低层结点。尽量让投影操作发生在限制的父结点里。投影操作将从（中间）结果元组里剔除用不着的属性，从而进一步减少需要在查询树中传递

① Y. E. Ioannidis, R. T. Ng, K. Shim, and T. Sellis. 1997. Parametric Query Optimization. *VLDB Journal* 6:132-151.

② 具体到这个例子，“高效率执行”不一定是最优方案。

③ 初次使用数据库的用户常用到技术。

的数据量。需要提醒大家注意的是，投影操作也许还会添加一些新属性——如果在它的父结点或更高层结点里有需要用到那些新属性的操作（如联结操作）。

(4) 把所有的联结操作放在被包含在join子句里的关系的投影或限制的交汇点^①。这是为了保证开销最大的操作（联结操作）处理的元组个数最少。有些联结操作可能需要优化器在查询树里临时创建一个中间结点来对某个子结点的结果元组进行排序。这些中间结点（我把它称为“工具结点”）可以根据联结操作的类型对有关元组进行排序或分组，它们可以极大地提高联结操作的性能。

注解 好的规则还有很多。上面列出的只是性能提升效果最好的几种。

把限制和选择操作尽可能推向低层结点的做法并不是人人赞成，Lee、Shih和Chen等人就反对这种做法^②。他们在工作中发现，在一定条件下，有些选取和投影操作的开销会比联结操作的开销更大。他们展示的一种基于图形理论的查询优化器可以更精确地对复杂的选取和投影操作做出更好的优化。不过，就一般情况而言，上面列出的规则已足以保证为绝大多数查询所生成的执行计划都足够好。

11.3 DBXP 查询优化器

虽然这些规则为查询树的生成提供了一套完整的操作，但未能给多向联结和索引以足够的重视。那些步骤被认为是基于开销的优化行为。因此，绝大多数启发式优化器会进行两个阶段的优化，第一阶段是生成一棵查询树，第二阶段是实施基于开销的优化策略。

DBXP优化器也将进行两个阶段的优化。第一阶段是使用一个启发式算法来调整树结点的位置；第二阶段是遍历查询树，为那些有索引可用的属性（字段）和关系（表）选用适当的访问方法。我将把基于开销的优化阶段留给大家作为练习。

11.3.1 测试设计

为一个启发式优化器创建一套完备的测试，需要编写大量的SQL代码才能覆盖所有可能的优化路径。简单地说，你将需要创建一个测试去测试有可能出现的一切查询，合法的和不合法的都得覆盖到。不过，实现这个启发式优化器只是DBXP引擎的第二个部分。上一章创建了一个基本的查询树内部表示并生成了执行方法存根。本章将创建一个优化器，但仍将无法执行那些查询。你可以继续使用这个执行存根来测试这个优化器，但这次不再给出查询结果，可以利用上一章里的代码把查询计划显示出来而不是查询结果。

按照这一思路，我设计了几个基本的查询命令来测试这个优化器是不是对查询进行了优化。下一章将讨论如何实现查询执行引擎的问题。代码清单11-1给出了DBXP查询优化器的一个示例测试文件。

① 但这可能导致联结操作无法使用索引。

② C. Lee, C. Shih和Y. Chen. 2001. A Graph-Theoretic Model for Optimizing Queries Involving Methods. *VLDB Journal* 9:327-343。

代码清单11-1 DBXP查询优化器的示例测试文件 (ExpertMySQLCh11.test)

```

#
# Sample test to test the SELECT DBXP optimizer
#

# Test 1:
SELECT DBXP * FROM staff;

# Test 2:
SELECT DBXP id FROM staff WHERE staff.id = '123456789';

# Test 3:
SELECT DBXP id, dir_name FROM staff, directorate
WHERE staff.dno = directorate.dnumber;

# Test 4:
SELECT DBXP * FROM staff JOIN tasking ON staff.id = tasking.id
WHERE staff.id = '123456789';

```

提示 这些例子所使用的数据库收录在本章的附录里。

当然，你完全可以在上面这个测试文件的基础上再添加一些自己的命令来测试我们将要添加的新代码。使用MySQL Test Suite工具来运行这个测试的详细步骤可以在本书的第4章里查到。

11.3.2 为 SELECT DBXP 命令生成存根

因为DBXP查询执行引擎要到下一章才能实现出来，我们现在只能对查询命令进行优化但无法执行。我们可以利用在上一章里实现的show_plan()函数（DBXP版EXPLAIN命令）去查看优化器是不是在工作以及工作得怎么样。为了添加这一功能，请打开sql_dbxp_parse.cc文件并把DBXP_select_command()方法修改成如代码清单11-2所示的样子。

代码清单11-2 为测试查询优化器生成存根

```

int DBXP_explain_select_command(THD *thd);

/*
Perform SELECT DBXP Command
SYNOPSIS
    DBXP_select_command()
    THD *thd             IN the current thread

DESCRIPTION
    This method executes the SELECT command using the query tree and optimizer.

RETURN VALUE
    Success = 0
    Failed = 1
*/

```

```
int DBXP_select_command(THD *thd)
{
    DEBUG_ENTER("DBXP_select_command");
    DBXP_explain_select_command(thd);
    DEBUG_RETURN(0);
}
```

这个修改将使有关代码去调用EXPLAIN命令代码而不是去执行查询命令。这可以让将要进行的测试返回有效的结果集（查询计划），从而使我们能在没有查询执行引擎部分的情况下对优化器进行测试。

注解 我在DBXP_select_command()方法的前面添加了一个函数声明。这使得有关代码无需使用头文件就可以调用DBXP_explain_select_command()方法了。

你还需要在DBXP_explain_select_command()方法里做一些修改。需要添加对heuristic_optimization()和cost_optimization()这两个新优化方法的调用。我将在接下来的几小节里对启发式优化做进一步的讨论。代码清单11-3给出了完成这些修改后的EXPLAIN代码。

代码清单11-3 修改EXPLAIN命令代码

```
/*
    Perform EXPLAIN command.

    SYNOPSIS
        DBXP_explain_select_command()
        THD *thd                IN the current thread

    DESCRIPTION
        This method executes the EXPLAIN SELECT command.

    RETURN VALUE
        Success = 0
        Failed = 1
*/
int DBXP_explain_select_command(THD *thd)
{
    bool res;
    select_result *result = thd->lex->result;

    DEBUG_ENTER("DBXP_explain_select_command");

    /* Prepare the tables (check access, locks) */
    res = check_table_access(thd, SELECT_ACL, thd->lex->query_tables, 0);
    if (res)
```

```

/* Create the query tree and optimize it */
Query_tree *qt = build_query_tree(thd, thd->lex,
    (TABLE_LIST*) thd->lex->select_lex.table_list.first);
qt->heuristic_optimization();
qt->cost_optimization();

/* create a field list for returning the query plan */
List<Item> field_list;

/* use the protocol class to communicate to client */
Protocol *protocol= thd->protocol;

/* write the field to the client */
field_list.push_back(new Item_empty_string("Execution Path",NAME_LEN));
if (protocol->send_fields(&field_list,
    Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
    DEBUG_RETURN(TRUE);
protocol->prepare_for_resend();

/* generate the query plan and send it to client */
show_plan(protocol, qt->root, qt->root, false);
send_eof(thd); /* end of file tells client no more data is coming */

/* unlock tables and cleanup memory */
mysql_unlock_read_tables(thd, thd->lock);
delete qt;
DEBUG_RETURN(0);
}

```

11.3.3 重要的 MySQL 结构和类

MySQL源代码里有许多重要的结构和类。你在前面各章的例子已经见过一些了。《MySQL内部手册》只对一些比较重要的结构和类进行了说明，没有把它们全部介绍过来。接下来的几个小节将对你与DBXP查询优化器（以及后面的查询执行引擎）打交道时会遇到的这些结构和类进行描述。这包括TABLE结构、Field类和几个常用的Item迭代器（Item类在第3章已经讨论过了）。

1. TABLE结构

在编写优化器代码时，将会用到的最重要的MySQL结构是TABLE结构。这个结构在/sql/tbale.h文件里被定义为st_table结构，在/sql/handler.h文件里又被再次定义为TABLE结构。绝大多数MySQL源代码都把这个结构称为TABLE。

这个结构非常重要，因为它包含着关于某给定表的所有信息，从一个指向某个存储引擎的指针到各种字段清单、键清单和用来存放中间结果（结果集）的临时缓冲区的清单，可以说应有尽有。这个结构相当大（MySQL里的重要结构都不小），但常见和常用的重要属性总是那几个。表11-1列出了TABLE结构的一些比较重要的属性。对TABLE结构的详细描述可以在handler.h文件里查到。

表11-1 TABLE结构的一些重要属性

属 性	说 明
file	一个指向存储引擎对象的引用指针
field	一个用来描述表各个字段的数组
fields	field数组里的元素（字段）个数
table_name	表的名字。这可以是假名或真名——这要取决于查询语句是如何引用它的
real_name	表的真名（不是假名）
path	当前表的*.frm文件的存放路径
record[]	用来在查询执行期间存放来自这个表的中间结果（结果集）的两个缓冲区
reclength	正被处理的记录的总长度（以字节为单位）
rec_buff_length	记录缓冲区的长度
keys	当前表里键的个数
next	一个指向表链表里的下一个表的指针
prev	一个指向表链表里的上一个表的指针

注解 MySQL 5.1版对path属性做了些修改，以允许使用不同的文件名和表名。

2. Field类

Field类包含着创建、赋值和处理某给定表里的字段（或属性）所需要的所有属性和方法。Field类的定义在/sql/field.h文件里，它的源代码在/sql/field.cc文件里。Field类其实是一个基类，各种字段类型都是从它派生出来的子类。这些子类的名字都是Field_XXX形式，它们散布在MySQL源代码里的多个地方。

因为Field类只是一个基类^①，所以它里面有许多方法都是为了让它的派生类加以覆写的（这些方法被定义为virtual类型）。尽管如此，Field类的许多派生类都有同样的基本属性和方法。表11-2列出了在与DBXP源代码打交道时会遇到的Field类的属性和方法。对Field类的详细描述可以在field.h文件里查到。

表11-2 Field类

属性/方法	说 明
ptr	一个指向记录缓冲区里的字段的指针
null_ptr	一个指向记录缓冲区里的一个字节的指针，用来表明这个字段是否可以包含空值NULL
table_name	这个字段所在的表的名字
field_name	这个字段的属性名
field_length	字段的长度。表明这个字段可以容纳多少个字节
is_null()	检查这个字段的值是不是空值NULL
move_field()	让内存里的字段指针指向另外一个地方
store()	一系列复用方法，用来把数据值存入各有关字段

① 它不是一个真正的抽象类，因为它包含若干定义在源代码中的方法。真正的抽象类将所有方法都定义为virtual，因此被用作接口而不是基类。

(续)

属性/方法	说 明
val_str()	读取字段的值, 把字段值返回为一个字符串
val_int()	读取字段的值, 把字段值返回为一个整数
result_type()	查出字段的数据类型
cmp()	把字段的当前值与给定的输入参数值进行比较, 并返回比较结果

3. 迭代器

在MySQL的源代码里有3种迭代器。我们在本书的第7章和第8章里曾经遇到过这些迭代器。迭代器可以简化创建和遍历一组对象的工作, 它们通常被实现为链表或数组。MySQL把迭代器实现为一些模板类, 可以通过一个输入参数来告诉它们我们想遍历哪种类型的数据。MySQL中的迭代器都是些链表, 但在行为上更象是队列和栈。接下来的几个小节描述了MySQL里的几个常用迭代器类。这些迭代器的定义都在/sql/sql_list.h头文件里。

□ template <> class List

List模板类被实现为一个队列或栈, 在队尾添加一个新元素的方法是push_back()方法, 在队首添加一个新元素的方法是push_front()方法。用来检索一个元素的方法是pop(), 用来删除一个元素的方法是remove()。你可以利用数据项的next属性来遍历整个队列, 但更常见的做法是先用List类创建一个链表(List<Item> item_list), 再用某个List_iterator类来快速遍历那个链表。这个类是从base_list类派生出来的, 后者的定义也在/sql/sql_list.h头文件里。

□ template <> class List_iterator

List_iterator类被实现为一个链表, 它使用重载的++操作符来遍历链表里的元素。用来检索一个元素的方法是ref(), 用来删除一个元素的方法是remove()。它还提供了一个rewind()方法, 用来直接跳到链表的第一个元素。这个类是从base_list类派生出来的, 后者的定义也在/sql/sql_list.h头文件里。

□ template <> class List_iterator_fast

除使用了一个经过优化的快速前序遍历算法以外, List_iterator_fast类与List_iterator类几乎完全一样。List_iterator_fast类被实现为一个链表, 它使用重载的++操作符来遍历链表里的元素。用来检索一个元素的方法是ref(), 用来删除一个元素的方法是remove()。这个类是从base_list类派生出来的, 后者的定义也在/sql/sql_list.h头文件里。

□ 用法示例

迭代器很容易使用。如果你想使用一个列表来处理一些数据项, 一个简单的List<Item_field>列表将是最佳选择。如果你想快速遍历一些字段的话, 可以用List_iterator<Item_field>或List_iterator_fast<Item_field>来创建一个列表迭代器。代码清单11-4给出了一些循环结构的例子。

代码清单11-4 迭代器用法示例

```
/* create a list and populate with some items */
List<Item> item_list;
item_list.push_back(new Item_int((int)32
                               join->select_lex->select_number));
```

```

item_list.push_back(new Item_string(join->select_lex->type,
    strlen(join->select_lex->type), cs));
item_list.push_back(new Item_string(message, strlen(message), cs));

../* start a basic list iterator to iterate through the item_list */
List_iterator<Item_field> item_list_it(*item_list);
/* control the iteration using an offset */
while ((curr_item= item_list_it++))
{
    /* do something */
}

../* start a fast list iterator to iterate through the item_list */
List_iterator_fast<Item_field> li(item_equal->fields);

/* control the iteration using an offset */
while ((item= li++))
{
    /* do something */
}

```

11.3.4 DBXP 辅助类

第9章曾经提到过两个DBXP引擎里的辅助类。设计这些类的目的是为了简化优化器的编程工作和提高代码的可读性。它们封装着一些MySQL类（和结构），并重复使用了MySQL源代码里的许多方法。

第一个辅助类是一个封装了查询命令中所用到的属性的类。这些属性在MySQL代码里是用Item或Item_field类来代表的，Attribute辅助类提供的通用访问接口可以让我们更方便地访问这些类。代码清单11-5给出了Attribute类的头文件。

代码清单11-5 Attribute类的头文件

```

class Attribute
{
public:
    Attribute(void);
    int remove_attribute(int num);
    Item *get_attribute(int num);
    int add_attribute(bool append, Item *new_item);
    int num_attributes();
    int index_of(char *table, char *value); /* find index of attr in list */
    int hide_attribute(Item *item, bool hide); /* remove from result set */
private:
    List<Item> attr_list;
    bool hidden[256]; /* used to indicate attributes not returned to client */
};

```

第二个辅助类是一个封装了查询命令中所用到的表达式的类。这些属性在MySQL代码里是用COND类来代表的，Expression辅助类为COND类提供了一个通用的（和简化的）接口。代码清单11-6给出了Expression类的头文件。

代码清单11-6 Expression类的头文件

```

struct expr_node
{
    COND      *left_op;
    COND      *operation;
    COND      *right_op;
    expr_node *next;
};

class Expression
{
public:
    Expression(void);
    int remove_expression(int num);
    expr_node *get_expression(int num);
    int add_expression(bool append, expr_node *new_item);
    int num_expressions();
    int index_of(char *table, char *value);
    int reduce_expressions(TABLE *table);
    bool has_table(char *table);
    int convert(COND *mysql_expr);
private:
    expr_node *root;
    int num_expr;
};

```

我使用了一个结构来容纳表达式，该结构里的元素依次对应着左操作数、操作符、右操作数。这比MySQL用一棵表达式树来代表一个表达式的做法要简单得多，让优化器的代码更易于阅读和理解。这个简单的技巧也让在一个交互式调试器里对条件进行求值的工作容易了许多。

注解 本书省略了这些辅助类的部分细节，因为它们都很简单，不过是为TABLE结构以及Item和Field类调用了一些MySQL方法而已。如果你想了解那些细节，可以从<http://www.apress.com>网站的Source Code主页下载本书的源代码；本章所有的源代码文件都收录在里面。

这些辅助类和辅助文件需要放入/sql子目录并添加到你的项目文件里。11.3.7小节将告诉你如何去做。

11.3.5 修改现有代码

为了实现DBXP优化器，还需要再做一些小的修改：添加一些代码来使用Attribute和Expression类。请打开query_tree.h头文件并按清单11-7修改有关代码。正如你看到的那样，我修改了where_expr和join_expr属性，让它们使用新的Expression类。类似的，我还修改了attributes属性，让它使用新的Attribute类。

代码清单11-7 修改查询树类 (query_tree.h文件)

```

struct query_node
{
    query_node();
    //query_node(const query_node &o);
    ~query_node();
    int          nodeid;
    int          parent_nodeid;
    bool         sub_query;
    int          child;
    query_node_type node_type;
    type_join    join_type;
    join_con_type join_cond;
    Expression   *where_expr;
    Expression   *join_expr;
    TABLE_LIST *relations[MAXNODETABLES];
    bool         preempt_pipeline;
    Attribute    *attributes;
    query_node   *left;
    query_node   *right;
};

```

还需要给查询树类添加几个新方法。因为篇幅的限制，就不逐一介绍每个方法和它的实现了。代码清单11-8是查询树类的其余定义语句，请把这些代码也添加到query_tree.h文件里去。

代码清单11-8 查询树类的新方法 (query_tree.h文件)

```

query_node *root;          //The ROOT node of the tree

Query_tree(void);
~Query_tree(void);
int heuristic_optimization();
int cost_optimization();
bool distinct;

private:
    bool h_opt;             //has query been optimized (rules)?
    bool c_opt;             //has query been optimized (cost)?

    int push_projections(query_node *QN, query_node *pNode);
    query_node *find_projection(query_node *QN);
    bool is_leaf(query_node *QN);
    bool has_relation(query_node *QN, char *Table);
    bool has_attribute(query_node *QN, Item *a);
    int del_attribute(query_node *QN, Item *a);

```

```

int prune_tree(query_node *prev, query_node *cur_node);
int balance_joins(query_node *QN);
int split_restrict_with_project(query_node *QN);
int split_restrict_with_join(query_node *QN);
int split_project_with_join(query_node *QN);
bool find_table_in_tree(query_node *QN, char *tbl);
bool find_table_in_expr(Expression *expr, char *tbl);
bool find_attr_in_expr(Expression *expr, char *tbl, char *value);
int apply_indexes(query_node *QN);
};

```

请注意，只有两个公用方法：heuristic_optimization()和cost_optimization()。还添加了一个名为distinct的公用属性，你可以利用这个属性来实现求异操作（详见本章末尾的练习题）。其余的方法是为编写优化器代码而准备的一些辅助方法。因为篇幅的限制，我将只解释几个最重要的，其他的留给读者自己去分析。

这些辅助类可以简化DBXP优化器的编程工作，但还需要由我们把它们用到适当的地方才能让它们发挥作用。现在，需要把一些新代码添加到负责把MySQL内部查询表示转换为DBXP查询树的build_query_tree()方法里去。请打开sql_dbxp.parse.cc文件并找到build_query_tree()方法，然后按照代码清单11-9进行修改。这将让这个方法用上新的Attribute类和Expression类。

代码清单11-9 修改build_query_tree()方法

```

/*
Build Query Tree

SYNOPSIS
    build_query_tree()
    THD *thd          IN the current thread
    LEX *lex          IN the pointer to the current parsed structure
    TABLE_LIST *tables IN the list of tables identified in the query

DESCRIPTION
    This method returns a converted MySQL internal representation (IR) of a
    query as a query_tree.

RETURN VALUE
    Success = Query_tree * -- the root of the new query tree.
    Failed = NULL
*/
Query_tree *build_query_tree(THD *thd, LEX *lex, TABLE_LIST *tables)
{
    DEBUG_ENTER("build_query_tree");
    Query_tree *qt = new Query_tree();
    Query_tree::query_node *qn =
        (Query_tree::query_node *)my_malloc(sizeof(Query_tree::query_node),
        MYF(MY_ZEROFILL | MY_WME));
    TABLE_LIST *table;
    int i = 0;
    Item *w;

```

```

int num_tables = 0;

/* create a new restrict node */
qn->parent_nodeid = -1;
qn->child = false;
qn->join_type = (Query_tree::type_join) 0;
qn->nodeid = 0;
qn->node_type = (Query_tree::query_node_type) 2;
qn->left = NULL;
qn->right = NULL;
qn->attributes = new Attribute();
qn->where_expr = new Expression();
qn->join_expr = new Expression();

if(lex->select_lex.options & SELECT_DISTINCT)
{
    //qt->set_distinct(true); /* placeholder for exercise */
}

/* Get the tables (relations) */
i = 0;
for(table = tables; table; table = table->next_local)
{
    num_tables++;
    qn->relations[i] = table;
    i++;
}

/* prepare the fields (find associated tables) for query */
list <Item> all_fields;
if (setup_wild(thd, tables, thd->lex->select_lex.item_list, &all_fields, 1))
    DEBUG_RETURN(NULL);
if (setup_fields(thd, lex->select_lex.ref_pointer_array,
    lex->select_lex.item_list, 1, &all_fields, 1))
    DEBUG_RETURN(NULL);
qt->result_fields = lex->select_lex.item_list;
/* get the attributes from the raw query */
w = lex->select_lex.item_list.pop();
while (w != 0)
{
    qn->attributes->add_attribute(true, w);
    w = lex->select_lex.item_list.pop();
}

/* get the joins from the raw query */
if (num_tables > 0) //indicates more than 1 table processed
    for(table = tables; table; table = table->next_local)
    {
        if (table->on_expr != 0)
            qn->join_expr->convert(thd, table->on_expr);
    }

```

```

    }

    /* get the expressions for the where clause */
    qn->where_expr->convert(thd, lex->select_lex.where);

    /* get the join conditions for the joins */
    qn->join_expr->get_join_expr(qn->where_expr);

    /* if there is a where clause, set node to restrict */
    if (qn->where_expr->num_expressions() > 0)
        qn->node_type = (Query_tree::query_node_type) 1;

    qt->root = qn;
    DEBUG_RETURN(qt);
}

```

接下来需要对头文件进行一些调整。当需要在某个头文件里引用另一个类的定义（或其他定义）时，通常会在这个头文件的开头用一条#include语句引入那个类的头文件；C++程序员肯定都很熟悉这种做法。MySQL编程指南不鼓励这样做，这是因为：如果被引入的头文件既包含有定义，又包含有代码实现，就会在编译时引起不必要的麻烦。

为了解决这个问题，需要把#include "mysql_priv.h"语句添加到源代码文件(*.cc)中，为头文件把它放到#include语句的前面。比如说，以下语句将出现在query_tree.cc文件的开头。

```

#include "mysql_priv.h"
#include "query_tree.h"

```

接下来，在query_tree.h文件的开头用下面的语句引入attribute.h和expression.h头文件。

```

#include "attribute.h"
#include "expression.h"

```

这将保证代码能够按照正确的顺序被编译而无需在mysql_priv.h文件里创建或修改任何东西。

注意 如果你在编译时遇到了奇怪的错误，请检查是不是把attribute.h、expression.h和query_tree.h头文件包括在编译里了；如果是，请去掉它们。编译器将根据我们刚才添加的指令自动包括这几个文件。

11.3.6 启发式优化器的细节

DBXP启发式优化器是根据前面描述的规则实现的，这些规则需要通过一系列函数方法来起作用。我把这些方法汇总在了表11-3里。

表11-3 DBXP启发式优化器里的启发式优化方法

方 法	说 明
split_restrict_with_join()	在查询树里搜索有一个限制（有表达式）和一个联结表达式的结点。它将把这样的结点一分为二：一个“限制”结点和一个“联结”结点
split_project_with_join()	在查询树里搜索有一个投影（有属性）和一个联结表达式的结点。它将把这样的结点一分为二：一个“投影”结点和一个“联结”结点

(续)

方 法	说 明
split_restrict_with_project()	在查询树里搜索有一个限制（有表达式）和一个投影（有属性）的结点。它将把这样的结点一分为二：一个“限制”结点和一个“投影”结点
find_restriction()	在查询树里搜索还不是一个叶结点的“限制”结点
push_restrictions()	把限制操作尽量下推到查询树的最低层结点，最好是叶结点。这个方法和find_restrictions()方法一起用在一个循环里（当再也找不到不是叶结点的“限制”结点时结束循环）
find_projection()	在查询树里搜索还不是一个叶结点的“投影”结点
push_projections()	把投影操作尽量下推到查询树的最低层结点，最好是叶结点。这个方法和find_projections()方法一起用在一个循环里（当再也找不到不是叶结点的“投影”结点或叶结点的父结点不再是一个限制操作时结束循环）
find_join()	在查询树里搜索“联结”结点
push_joins()	把联结操作下推到查询树的低层结点，让它成为它所联结的“限制”和/或“投影”结点的直接父结点
prune_tree()	在查询树里搜索因优化而不再有用的结点（既没有属性或表达式，也不是联结或排序操作）并删掉它们

DBXP启发式优化器的源代码有非常高的可读性。代码清单11-10给出了heuristic_optimization()方法的源代码实现。

代码清单11-10 DBXP优化器：heuristic_optimization()方法

```

/*
Perform heuristic optimization

SYNOPSIS
    heuristic_optimization()

DESCRIPTION
    This method performs heuristic optimization on the query tree. The
    operation is destructive in that it rearranges the original tree.

RETURN VALUE
    . Success = 0
    . Failed = 1
*/
int Query_tree::heuristic_optimization()
{
    DEBUG_ENTER("heuristic_optimization");
    query_node      *pNode;
    query_node      *nNode;

    h_opt = true;
    /*
    First, we have to correct the situation where restrict and
    project are grouped together in the same node.
    */

```

```

split_restrict_with_join(root);
split_project_with_join(root);
split_restrict_with_project(root);

/*
Find a node with restrictions and push down the tree using
a recursive call. continue until you get the same node twice.
This means that the node cannot be pushed down any further.
*/
pNode = find_restriction(root);
while(pNode != 0)
{
    push_restrictions(root, pNode);
    nNode = find_restriction(root);
/*
If a node is found, save a reference to it unless it is
either the same node as the last node found or
it is a leaf node. This is done so that we can ensure we
continue searching down the tree visiting each node once.
*/
if(nNode != 0)
{
    if(nNode->nodeid == pNode->nodeid)
        pNode = 0;
    else if(is_leaf(nNode))
        pNode = 0;
    else
        pNode = nNode;
}
}

/*
Find a node with projections and push down the tree using
a recursive call. Continue until you get the same node twice.
This means that the node cannot be pushed down any further.
*/
pNode = find_projection(root);
while(pNode != 0)
{
    push_projections(root, pNode);
    nNode = find_projection(root);
/*
If a node is found, save a reference to it unless it is
either the same node as the last node found or
it is a leaf node. This is done so that we can ensure we
continue searching down the tree visiting each node once.
*/
if(nNode != 0)
{
    if(nNode->nodeid == pNode->nodeid)

```

```

        pNode = 0;
    else if(is_leaf(nNode))
        pNode = 0;
    else
        pNode = nNode;
    }
}

/*
Find a join node and push it down the tree using
a recursive call. Continue until you get the same node twice.
This means that the node cannot be pushed down any further.
*/
pNode = find_join(root);
while(pNode != 0)
{
    push_joins(root, pNode);
    nNode = find_join(root);
    /*
    If a node is found, save a reference to it unless it is
    either the same node as the last node found or
    it is a leaf node. This is done so that we can ensure we
    continue searching down the tree visiting each node once.
    */
    if(nNode != 0)
    {
        if(nNode->nodeid == pNode->nodeid)
            pNode = 0;
        else if(is_leaf(nNode))
            pNode = 0;
        else
            pNode = nNode;
    }
    else
        pNode = nNode;
}

/*
Prune the tree of "blank" nodes
Blank Nodes are:
1) projections without attributes that have at least 1 child
2) restrictions without expressions
BUT...Can't delete a node that has TWO children!
*/
prune_tree(0, root);

/*
Lastly, check to see if this has the DISTINCT option.
If so, create a new node that is a DISTINCT operation.
*/

```

```

if(distinct && (root->node_type != qntDistinct))
{
    int i;
    pNode = (query_node*)my_malloc(sizeof(query_node),
        MYF(MY_ZEROFILL | MY_WME));
    pNode->sub_query = 0;
    pNode->attributes = 0;
    pNode->join_cond = jcUN; /* (join_con_type) 0; */
    pNode->join_type = jnUNKNOWN; /* (type_join) 0; */
    pNode->left = root;
    pNode->right = 0;
    for(i = 0; i < MAXNODETABLES; i++)
        pNode->relations[i] = NULL;
    pNode->nodeid = 90125;
    pNode->child = LEFTCHILD;
    root->parent_nodeid = 90125;
    root->child = LEFTCHILD;
    pNode->parent_nodeid = -1;
    pNode->node_type = qntDistinct;
    pNode->attributes = new Attribute();
    pNode->where_expr = new Expression();
    pNode->join_expr = new Expression();
    root = pNode;
}
DEBUG_RETURN(0);
}

```

请注意那几个用来搜索“限制”、“投影”和“联结”结点的循环。它们使用了一种前序遍历算法来遍历查询树并根据规则做出处理，直到再也找不到违反规则的“坏”结点为止。

在接下来的几份代码清单里，你将看到heuristic_optimization()方法里的几个主要方法的源代码。为了节约篇幅，我省略了一些不太重要的辅助方法，它们大都只是MySQL结构和类方法的简单抽象而已。如果想了解其他辅助方法的细节，你应该下载本章的源代码文件。

split_restrict_with_join()方法在查询树里搜索带有where表达式的“联结”结点（这意味着它包含限制和联结两个操作）并把它们一分为二：一个“限制”结点和一个“联结”结点。代码清单11-11给出了这个方法的源代码。

代码清单11-11 split_restrict_with_join()方法

```

/*
Split restrictions that have joins.

SYNOPSIS
split_restrict_with_join()
query_node *QN IN the node to operate on

DESCRIPTION
This method looks for joins that have where expressions (thus are both
joins and restrictions) and breaks them into two nodes.

```


NOTES

This is a RECURSIVE method!

RETURN VALUE

Success = 0

Failed = 1

*/

```
int Query_tree::split_restrict_with_join(query_node *QN)
```

```
{
```

```
    int j = 0;
```

```
    int i = 0;
```

```
    DEBUG_ENTER("split_restrict_with_join");
```

```
    if(QN != 0)
```

```
    {
```

```
        if(((QN->join_expr->num_expressions() > 0) &&
```

```
            (QN->where_expr->num_expressions() > 0)) &&
```

```
            ((QN->node_type == qntJoin) || (QN->node_type == qntRestrict))))
```

```
        {
```

```
            bool isleft = true;
```

```
            /*
```

Create a new node and:

1) Move the where expressions to the new node.

2) Set the new node's children = current node children

3) Set the new node's relations = current node relations.

4) Set current node's left or right child = new node;

5) Set new node's id = current id + 200;

6) set parent id, etc.

7) determine which table needs to be used for the restrict node.

```
            */
```

```
            query_node *new_node = (query_node*)my_malloc(sizeof(query_node),
                                                            MYF(MY_ZEROFILL | MY_WME));
```

```
            new_node->node_type = qntRestrict;
```

```
            new_node->parent_nodeid = QN->nodeid;
```

```
            new_node->nodeid = QN->nodeid + 200;
```

```
            new_node->where_expr = QN->where_expr;
```

```
            new_node->join_expr = new Expression();
```

```
            QN->where_expr = new Expression();
```

```
            /*
```

Loop through tables and move table that matches to the new node

```
            */
```

```
            for(i = 0; i < MAXNODETABLES; i++)
```

```
            {
```

```
                if (QN->relations[i] != NULL)
```

```
                {
```

```
                    if (find_table_in_expr(new_node->where_expr,
                                            QN->relations[i]->table_name))
```

```

    {
        new_node->relations[j] = QN->relations[i];
        j++;
        if (i != 0)
            isleft = false;
        QN->relations[i] = NULL;
    }
}

/* set children to point to balance of tree */
new_node->right = 0;
if (isleft)
{
    new_node->child = LEFTCHILD;
    new_node->left = QN->left;
    QN->left = new_node;
}
else
{
    new_node->child = RIGHTCHILD;
    new_node->left = QN->right;
    QN->right = new_node;
}
if (new_node->left)
    new_node->left->parent_nodeid = new_node->nodeid;
j = QN->attributes->num_attributes();
if ((QN->node_type == qntJoin) && (j > 0))
{
    Attribute *attribs = 0;
    Item *attr;
    int ii = 0;
    int jj = 0;
    if ((QN->attributes->num_attributes() == 1) &&
        (strcmp("(",
            ((Field *)QN->attributes->get_attribute(0))->field_name) == 0))
    {
        new_node->attributes = new Attribute();
        new_node->attributes->add_attribute(j,
            QN->attributes->get_attribute(0));
    }
    else
    {
        attribs = new Attribute();
        for (i = 0; i < (int)new_node->relations[0]->table->s->fields; i++)
        {
            Item *f = (Item *)new_node->relations[0]->table->field[i];
            attribs->add_attribute(true, (Item *)f);
        }
        j = attribs->num_attributes();
    }
}

```

```

new_node->attributes = new Attribute();
for (i = 0; i < j; i++)
{
    attr = attrbs->get_attribute(i);
    jj = QN->attributes->index_of(
        (char *)((Field *)attr)->table->s->table_name.str,
        (char *)((Field *)attr)->field_name);
    if (jj > -1)
    {
        new_node->attributes->add_attribute(ii, attr);
        ii++;
        QN->attributes->remove_attribute(jj);
    }
    else if (find_attr_in_expr(QN->join_expr,
        (char *)((Field *)attr)->table->s->table_name.str,
        (char *)((Field *)attr)->field_name))
    {
        new_node->attributes->add_attribute(ii, attr);
        new_node->attributes->hide_attribute(attr, true);
        ii++;
    }
}
}
}
else
{
    QN->node_type = qntJoin;
    new_node->attributes = new Attribute();
}
}
split_restrict_with_join(QN->left);
split_restrict_with_join(QN->right);
}
DEBUG_RETURN(0);
}

```

split_project_with_join()方法在查询树里搜索带有属性(字段)的“联结”结点(这意味着它包含投影和联结两个操作)并把它们一分为二:一个“投影”结点和一个“联结”结点。代码清单11-12给出了这个方法的源代码。

代码清单11-12 split_project_with_join()方法

```

/*
Split projections that have joins.

SYNOPSIS
split_project_with_join()
query_node *QN IN the node to operate on
DESCRIPTION
This method looks for joins that have attributes (thus are both

```

joins and projections) and breaks them into two nodes.

NOTES

This is a RECURSIVE method!

RETURN VALUE

Success = 0

Failed = 1

```

*/
int Query_tree::split_project_with_join(query_node *QN)
{
    int j = 0;
    int i;

    DEBUG_ENTER("split_project_with_join");
    if(QN != 0)
    {
        if((QN->join_expr->num_expressions() > 0) &&
            ((QN->node_type == qntJoin) || (QN->node_type == qntProject)))
        {
            /*
             Create a new node and:
             1) Move the where expressions to the new node.
             2) Set the new node's children = current node children
             3) Set the new node's relations = current node relations.
             4) Set current node's left or right child = new node;
             5) Set new node's id = current id + 300;
             6) set parent id, etc.
            */
            QN->node_type = qntJoin;
            if (QN->left == 0)
            {
                query_node *new_node = (query_node*)my_malloc(sizeof(query_node),
                                                                MYF(MY_ZEROFILL | MY_WME));
                new_node->node_type = qntProject;
                new_node->parent_nodeid = QN->nodeid;
                new_node->nodeid = QN->nodeid + 300;
                for(i = 0; i < MAXNODETABLES; i++)
                    new_node->relations[i] = 0;
                new_node->relations[0] = QN->relations[0];
                QN->relations[0] = 0;
                new_node->left = QN->left;
                QN->left = new_node;
                new_node->right = 0;
                new_node->child = LEFTCHILD;
                if (new_node->left != 0)
                    new_node->left->parent_nodeid = new_node->nodeid;
                j = QN->attributes->num_attributes();
                new_node->attributes = new Attribute();
                new_node->where_expr = new Expression();
            }
        }
    }
}

```



```

new_node->join_expr = new Expression();
if ((j == 1) &&
    (strcasecmp("?", QN->attributes->get_attribute(0)->name) == 0))
{
    new_node->attributes = new Attribute();
    new_node->attributes->add_attribute(j, QN->attributes->get_attribute(0));
    if (QN->right != 0)
        QN->attributes->remove_attribute(0);
}
else if (j > 0)
{
    Attribute *attribs = 0;
    Item *attr;
    int ii = 0;
    int jj = 0;
    attribs = new Attribute();
    for (i = 0; i < (int)new_node->relations[0]->table->s->fields; i++)
    {
        Field *f = new_node->relations[0]->table->field[i];
        attribs->add_attribute(true, (Item *)f);
    }
    j = attribs->num_attributes();
    for (i = 0; i < j; i++)
    {
        attr = attribs->get_attribute(i);
        jj = QN->attributes->index_of(
            (char *)((Field *)attr)->table->s->table_name.str,
            (char *)((Field *)attr)->field_name);
        if (jj > -1)
        {
            new_node->attributes->add_attribute(ii, attr);
            ii++;
            QN->attributes->remove_attribute(jj);
        }
        else if (find_attr_in_expr(QN->join_expr,
            (char *)((Field *)attr)->table->s->table_name.str,
            (char *)((Field *)attr)->field_name))
        {
            new_node->attributes->add_attribute(ii, attr);
            new_node->attributes->hide_attribute(attr, true);
            ii++;
        }
    }
}
if (QN->right == 0)
{
    query_node *new_node = (query_node*)my_malloc(sizeof(query_node),
        MYF(MY_ZEROFILL | MY_WME));

```

```

new_node->node_type = qntProject;
new_node->parent_nodeid = QN->nodeid;
new_node->nodeid = QN->nodeid + 400;
for(i = 0; i < MAXNODETABLES; i++)
    new_node->relations[0] = 0;
new_node->relations[0] = QN->relations[1];
QN->relations[1] = 0;
new_node->left = QN->right;
QN->right = new_node;
new_node->right = 0;
new_node->child = RIGHTCHILD;
if (new_node->left != 0)
    new_node->left->parent_nodeid = new_node->nodeid;
j = QN->attributes->num_attributes();
new_node->attributes = new Attribute();
new_node->where_expr = new Expression();
new_node->join_expr = new Expression();
if ((j == 1) &&
    (strcasecmp("?", (char *)QN->attributes->get_attribute(0)->name) == 0))
{
    new_node->attributes = new Attribute();
    new_node->attributes->add_attribute(j, QN->attributes->get_attribute(0));
    QN->attributes->remove_attribute(0);
}
else if (j > 0)
{
    Attribute *attribs = 0;
    Item *attr;
    int ii = 0;
    int jj = 0;
    attribs = new Attribute();
    for (i = 0; i < (int)new_node->relations[0]->table->s->fields; i++)
    {
        Field *f = new_node->relations[0]->table->field[i];
        attribs->add_attribute(true, (Item *)f);
    }
    j = attribs->num_attributes();
    new_node->attributes = new Attribute();
    for (i = 0; i < j; i++)
    {
        attr = attribs->get_attribute(i);
        jj = QN->attributes->index_of(
            (char *)((Field *)attr)->table->s->table_name.str,
            (char *)((Field *)attr)->field_name);
        if (jj > -1)
        {
            new_node->attributes->add_attribute(ii, attr);
            ii++;
            QN->attributes->remove_attribute(jj);
        }
    }
}

```

```

        else if (find_attr_in_expr(QN->join_expr,
            (char *)((Field *)attr)->table->s->table_name.str,
            (char *)((Field *)attr)->field_name))
        {
            new_node->attributes->add_attribute(ii, attr);
            new_node->attributes->hide_attribute(attr, true);
            ii++;
        }
    }
}
split_project_with_join(QN->left);
split_project_with_join(QN->right);
}
DEBUG_RETURN(0);
}

```

split_restrict_with_project()方法在查询树里搜索带有属性（字段）的“限制”结点（这意味着它包含限制和投影两个操作）并把它们一分为二：一个“限制”结点和一个“投影”结点。代码清单11-13给出了这个方法的源代码。

代码清单11-13 split_restrict_with_project()方法

```

/*
  Split restrictions that have attributes (projections).

  SYNOPSIS
    split_restrict_with_project()
    query_node *QN IN the node to operate on

  DESCRIPTION
    This method looks for restrictions that have attributes (thus are both
    projections and restrictions) and breaks them into two nodes.

  NOTES
    This is a RECURSIVE method!
  RETURN VALUE
    Success = 0
    Failed = 1
*/
int Query_tree::split_restrict_with_project(query_node *QN)
{
    DEBUG_ENTER("split_restrict_with_project");
    if(QN != 0)
    {
        if(((QN->attributes->num_attributes() > 0) &&
            (QN->where_expr->num_expressions() > 0)) &&
            ((QN->node_type == qntProject) || (QN->node_type == qntRestrict)))

```

```

{
    /*
    Create a new node and:
    1) Move the expressions to the new node.
    2) Set the new node's children = current node children
    3) Set the new node's relations = current node relations.
    4) Set current node's left child = new node;
    5) Set new node's id = current id + 1000;
    6) set parent id, etc.
    */
    query_node *new_node = (query_node*)my_malloc(sizeof(query_node),
                                                MYF(MY_ZEROFILL | MY_WME));
    new_node->child = LEFTCHILD;
    new_node->node_type = qntRestrict;
    if(new_node->node_type == qntJoin)
    {
        new_node->join_cond = QN->join_cond;
        new_node->join_type = QN->join_type;
    }
    QN->node_type = qntProject;
    new_node->attributes = new Attribute();
    new_node->where_expr = QN->where_expr;
    new_node->join_expr = new Expression();
    QN->where_expr = new Expression();
    new_node->left = QN->left;
    new_node->right = QN->right;
    new_node->parent_nodeid = QN->nodeid;
    new_node->nodeid = QN->nodeid + 1000;
    if(new_node->left)
        new_node->left->parent_nodeid = new_node->nodeid;
    if(new_node->right)
        new_node->right->parent_nodeid = new_node->nodeid;
    for(int i = 0; i < MAXNODETABLES; i++)
    {
        new_node->relations[i] = QN->relations[i];
        QN->relations[i] = NULL;
    }
    QN->left = new_node;
    QN->right = 0;
}
split_restrict_with_project(QN->left);
split_restrict_with_project(QN->right);
}
DEBUG_RETURN(0);
}

```

find_restriction()方法从起始结点(QN)开始,在查询树里寻找下一个“限制”结点。如果找到一个“限制”结点,这个方法将返回一个指向该结点的指针。否则,这个方法将返回空值NULL。代码清单11-14给出了这个方法的源代码。

代码清单11-14 find_restriction()方法

```

/*
Find a restriction in the subtree.

SYNOPSIS
    find_restriction()
    query_node *QN IN the node to operate on

DESCRIPTION
    This method looks for a node containing a restriction and returns the node
    pointer.

NOTES
    This is a RECURSIVE method!
    This finds the first restriction and is biased to the left tree.

RETURN VALUE
    Success = query_node * the node located
    Failed = NULL
*/
Query_tree::query_node *Query_tree::find_restriction(query_node *QN)
{
    DEBUG_ENTER("find_restriction");
    query_node *N;

    N = 0;
    if(QN != 0)
    {
        /*
        A restriction is a node marked as restrict and
        has at least one expression
        */
        if (QN->where_expr->num_expressions() > 0)
            N = QN;
        else
        {
            N = find_restriction(QN->left);
            if(N == 0)
                N = find_restriction(QN->right);
        }
    }
    DEBUG_RETURN(N);
}

```

push_restrictions()方法从起始结点 (QN) 开始搜索查询树, 它将把“限制”结点 (pNode) 一直下推到能让这个限制操作最早完成的最低层结点为止。代码清单11-15给出了这个方法的源代码。

代码清单11-15 push_restrictions()方法

```

/*
  Push restrictions down the tree.

  SYNOPSIS
    push_restrictions()
    query_node *QN IN the node to operate on
    query_node *pNode IN the node containing the restriction attributes

  DESCRIPTION
    This method looks for restrictions and pushes them down the tree to nodes
    that contain the relations specified.

  NOTES
    This is a RECURSIVE method!
    This finds the first restriction and is biased to the left tree.

  RETURN VALUE
    Success = 0
    Failed = 1
*/
int Query_tree::push_restrictions(query_node *QN, query_node *pNode)
{
    query_node      *NewQN;

    DEBUG_ENTER("push_restrictions");
    if((QN != 0) && (pNode != 0) && (pNode->left != 0))
    {
        /*
        Conditions:
        1) QN is a join node
        2) QN is a project node
        3) QN is a restrict node
        4) All other nodes types are ignored.

        Methods:
        1) if join or project and the children are not already restrictions
           add a new node and put where clause in new node else
           see if you can combine the child node and this one
        2) if the node has the table and it is a join,
           create a new node below it and push the restriction
           to that node.
        4) if the node is a restriction and has the table,
           just add the expression to the node's expression list
        */

        /* if projection, move node down tree */
        if((QN->nodeid != pNode->nodeid) && (QN->node_type == qntProject))
        {

```

```

if (QN->left != 0)
{
    QN->left = (query_node*)my_malloc(sizeof(query_node),
        MYF(MY_ZEROFILL | MY_WME));
    NewQN = QN->left;
    NewQN->left = 0;
}
else
{
    NewQN = QN->left;
    QN->left = (query_node*)my_malloc(sizeof(query_node),
        MYF(MY_ZEROFILL | MY_WME));
    QN->left->left = NewQN;
    NewQN = QN->left;
}
NewQN->sub_query = 0;
NewQN->join_cond = jcUN; /* (join_con_type) 0; */
NewQN->join_type = jnUNKNOWN; /* (type_join) 0; */
NewQN->right = 0;
for(long i = 0; i < MAXNODETABLES; i++)
    NewQN->relations[i] = 0;
NewQN->nodeid = QN->nodeid + 1;
NewQN->parent_nodeid = QN->nodeid;
NewQN->node_type = qntRestrict;
NewQN->attributes = new Attribute();
NewQN->where_expr = new Expression();
NewQN->join_expr = new Expression();
if (pNode->relations[0])
    NewQN->where_expr->reduce_expressions(pNode->relations[0]->table);
if ((QN->relations[0] != NULL) && (QN->relations[0] == pNode->relations[0]))
    if (QN->relations[0])
        if (find_table_in_expr(pNode->where_expr, QN->relations[0]->table_name))
        {
            NewQN->relations[0] = QN->relations[0];
            QN->relations[0] = 0;
        }
    else
    {
        if (pNode->relations[0])
            if (find_table_in_tree(QN->left, pNode->relations[0]->table_name))
                NewQN->relations[0] = 0;
        pNode->where_expr = NULL;
        pNode->relations[0] = 0;
    }
}
/* if join, move restrict node down tree */
else if((QN->nodeid != pNode->nodeid) &&
    ((QN->left == 0) || (QN->right == 0)) &&
    (QN->node_type == qntJoin))

```

```

{
    if(QN->relations[0] != 0)
    {
        QN->left = (query_node*)my_malloc(sizeof(query_node),
            MYF(MY_ZEROFILL | MY_WME));
        NewQN = QN->left;
        NewQN->sub_query = 0;
        NewQN->join_cond = jcUN; /* (join_con_type) 0; */
        NewQN->join_type = jnUNKNOWN; /* (type_join) 0; */
        NewQN->left = 0;
        NewQN->right = 0;
        for(long i = 0; i < MAXNODETABLES; i++)
            NewQN->relations[i] = 0;
        NewQN->nodeid = QN->nodeid + 1;
        NewQN->parent_nodeid = QN->nodeid;
        NewQN->node_type = qntRestrict;
        NewQN->attributes = new Attribute();
        NewQN->where_expr = new Expression();
        NewQN->join_expr = new Expression();
        NewQN->relations[0] = QN->relations[0];
        QN->relations[0] = 0;
        if (pNode->relations[0])
            NewQN->where_expr->reduce_expressions(pNode->relations[0]->table);
    }

    else if(QN->relations[1] != 0)
    {
        QN->right = (query_node*)my_malloc(sizeof(query_node),
            MYF(MY_ZEROFILL | MY_WME));
        NewQN = QN->right;
        NewQN->sub_query = 0;
        NewQN->join_cond = jcUN; /* (join_con_type) 0; */
        NewQN->join_type = jnUNKNOWN; /* (type_join) 0; */
        NewQN->left = 0;
        NewQN->right = 0;
        for(long i = 0; i < MAXNODETABLES; i++)
            NewQN->relations[i] = 0;
    }
    NewQN->nodeid = QN->nodeid + 1;
    NewQN->parent_nodeid = QN->nodeid;
    NewQN->node_type = qntRestrict;
    NewQN->attributes = new Attribute();
    NewQN->where_expr = new Expression();
    NewQN->join_expr = new Expression();
    NewQN->relations[0] = QN->relations[1];
    QN->relations[1] = 0;
    NewQN->where_expr->reduce_expressions(pNode->relations[0]->table);
}
push_restrictions(QN->left, pNode);
push_restrictions(QN->right, pNode);

```



```

    }
    DEBUG_RETURN(0);
}

```

find_projection()方法从起始结点(QN)开始,在查询树里寻找下一个“投影”结点。如果找到一个“投影”结点,这个方法将返回一个指向该结点的指针;否则,这个方法将返回空值NULL。代码清单11-16给出了这个方法的源代码。

代码清单11-16 find_projection()方法

```

/*
Find a projection in the tree

SYNOPSIS
    find_projection()
    query_node *QN IN the node to operate on

DESCRIPTION
    This method looks for a node containing a projection and returns the node
    pointer.

NOTES
    This finds the first projection and is biased to the left tree.
    This is a RECURSIVE method!

RETURN VALUE
    Success = query_node * the node located or NULL for not found
    Failed = NULL
*/
Query_tree::query_node *Query_tree::find_projection(query_node *QN)
{
    DEBUG_ENTER("find_projection");
    query_node *N;

    N = 0;
    if(QN != 0)
    {
        /*
        A projection is a node marked as project and
        has at least one attribute
        */
        if((QN->node_type == qntProject) &&
            (QN->attributes != 0))
            N = QN;
        else
        {
            N = find_projection(QN->left);
        }
    }
}

```

```

    DBUG_RETURN(N);
}

```

push_projections()方法从起始结点(QN)开始搜索查询树,它将把“投影”结点(pNode)一直下推到能让这个投影操作最早完成的最低层结点为止。代码清单11-17给出了这个方法的源代码。

代码清单11-17 push_projections()方法

```

/*
  Push projections down the tree.

  SYNOPSIS
    push_projections()
    query_node *QN IN the node to operate on
    query_node *pNode IN the node containing the projection attributes
  DESCRIPTION
    This method looks for projections and pushes them down the tree to nodes
    that contain the relations specified.

  NOTES
    This is a RECURSIVE method!

  RETURN VALUE
    Success = 0
    Failed = 1
*/
int Query_tree::push_projections(query_node *QN, query_node *pNode)
{
    DBUG_ENTER("push_projections");
    Item * a;
    int i;
    int j;

    if((QN != 0) && (pNode != 0))
    {
        if((QN->nodeid != pNode->nodeid) &&
            (QN->node_type == qntProject))
        {
            i = 0;
            j = QN->attributes->num_attributes();

            /* move attributes to new node */
            while(i < j)
            {
                a = QN->attributes->get_attribute(i);
                if(has_relation(QN,
                    (char *)((Field *)a->table->s->table_name.str))
                {
                    if(!has_attribute(QN, a))
                        insert_attribute(QN, a);
                }
            }
        }
    }
}

```

```

        del_attribute(pNode, a);
    }
    i++;
}
}
if(pNode->attributes->num_attributes() != 0)
{
    push_projections(QN->left, pNode);
    push_projections(QN->right, pNode);
}
}
DEBUG_RETURN(0);
}

```

find_join()方法从起始结点(QN)开始在查询树里寻找下一个“联结”结点。如果找到一个“联结”结点，这个方法将返回一个指向该结点的指针；否则，这个方法将返回空值NULL。代码清单11-18给出了这个方法的源代码。

代码清单11-18 find_join()方法

```

/*
Find a join in the subtree.

SYNOPSIS
find_restriction()
query_node *QN IN the node to operate on

DESCRIPTION
This method looks for a node containing a join and returns the
node pointer.

NOTES
This is a RECURSIVE method!
This finds the first restriction and is biased to the left tree.

RETURN VALUE
Success = query_node * the node located
Failed = NULL
*/
Query_tree::query_node *Query_tree::find_join(query_node *QN)
{
    DEBUG_ENTER("find_join");
    query_node *N;
    N = 0;

    if(QN != 0)
    {
        /*
        if this is a restrict node or a restrict node with
        at least one expression it could be an unprocessed join

```

```

        because the default node type is restrict
    */
    if(((QN->node_type == qntRestrict) ||
        (QN->node_type == qntRestrict)) && (QN->join_expr->num_expressions() > 0))
        N = QN;
    else
    {
        N = find_join(QN->left);
        if(N == 0)
            N = find_join(QN->right);
    }
}
DEBUG_RETURN(N);
}

```

push_joins()方法从起始结点 (QN) 开始搜索查询树, 它将把“联结”结点 (pNode) 一直下推到能让这个联结操作最早完成的最低层结点为止。代码清单11-19给出了这个方法的源代码。

代码清单11-19 push_joins()方法

```

/*
    Push joins down the tree.

    SYNOPSIS
        push_restrictions()
        query_node *QN IN the node to operate on
        query_node *pNode IN the node containing the join

    DESCRIPTION
        This method looks for theta joins and pushes them down the tree to the
        parent of two nodes that contain the relations specified.

    NOTES
        This is a RECURSIVE method!

    RETURN VALUE
        Success = 0
        Failed = 1
*/
int Query_tree::push_joins(query_node *QN, query_node *pNode)
{
    DEBUG_ENTER("push_joins");
    COND *lField;
    COND *rField;
    expr_node *node;

    if(!pNode->join_expr)
        DEBUG_RETURN(0);
    node = pNode->join_expr->get_expression(0);
    if (!node)

```



```

    DEBUG_RETURN(0);
    lField = node->left_op;
    rField = node->right_op;

    /* Node must have expressions and not be null */
    if((QN != NULL) && (pNode != NULL) &&
        (pNode->join_expr->num_expressions() > 0))
    {
        /* check to see if tables in join condition exist */
        if((QN->nodeid != pNode->nodeid) &&
            (QN->node_type == qntJoin) &&
            QN->join_expr->num_expressions() == 0 &&
            ((has_relation(QN->left,
                (char *)((Field *)lField)->table->s->table_name.str) &&
                has_relation(QN->right,
                (char *)((Field *)rField)->table->s->table_name.str)) ||
            (has_relation(QN->left,
                (char *)((Field *)rField)->table->s->table_name.str) &&
                has_relation(QN->right,
                (char *)((Field *)lField)->table->s->table_name.str))))
        {
            /* move the expression */
            QN->join_expr = pNode->join_expr;
            pNode->join_expr = new Expression();
            QN->join_type = jnINNER;
            QN->join_cond = jcON;
        }
        push_joins(QN->left, pNode);
        push_joins(QN->right, pNode);
    }
    DEBUG_RETURN(0);
}

```

prune_tree()方法在查询树里搜索因为进行启发式优化而不再有用的空白结点并删掉它们。代码清单11-20给出了这个方法的源代码。

代码清单11-20 prune_tree()方法

```

/*
    Prune the tree of dead limbs.

    SYNOPSIS
        prune_tree()
        query_node *prev IN the previous node (parent)
        query_node *cur_node IN the current node pointer (used to delete).

    DESCRIPTION
        This method looks for blank nodes that are a result of performing
        heuristic optimization on the tree and deletes them.

    NOTES
        This is a RECURSIVE method!

```

```

RETURN VALUE
    Success = 0
    Failed = 1
*/
int Query_tree::prune_tree(query_node *prev, query_node *cur_node)
{
    DEBUG_ENTER("prune_tree");
    if(cur_node != 0)
    {
        /*
         Blank Nodes are 1) projections without attributes
         that have at least 1 child, or 2) restrictions
         without expressions
        */
        if((((cur_node->node_type == qntProject) &&
            (cur_node->attributes->num_attributes() == 0)) ||
            ((cur_node->node_type == qntRestrict) &&
            (cur_node->where_expr->num_expressions() == 0))) &&
            ((cur_node->left == 0) || (cur_node->right == 0)))
        {
            /*
             Redirect the pointers for the nodes above and
             below this node in the tree.
            */
            if(prev == 0)
            {
                if(cur_node->left == 0)
                {
                    cur_node->right->parent_nodeid = -1;
                    root = cur_node->right;
                }
                else
                {
                    cur_node->left->parent_nodeid = -1;
                    root = cur_node->left;
                }
                my_free((gptr)cur_node, MYF(0));
                cur_node = root;
            }
            else
            {
                if(prev->left == cur_node)
                {
                    if(cur_node->left == 0)
                    {
                        prev->left = cur_node->right;
                        if (cur_node->right != NULL)
                            cur_node->right->parent_nodeid = prev->nodeid;
                    }
                    else

```

```

    {
        prev->left = cur_node->left;
        if (cur_node->left != NULL)
            cur_node->left->parent_nodeid = prev->nodeid;
    }
    my_free((gptr)cur_node, MYF(0));
    cur_node = prev->left;
}
else
{
    if (cur_node->left == 0)
    {
        prev->right = cur_node->right;
        if (cur_node->right != NULL)
            cur_node->right->parent_nodeid = prev->nodeid;
    }
    else
    {
        prev->right = cur_node->left;
        if (cur_node->left != NULL)
            cur_node->left->parent_nodeid = prev->nodeid;
    }
    my_free((gptr)cur_node, MYF(0));
    cur_node = prev->right;
}
}
prune_tree(prev, cur_node);
}
else
{
    prune_tree(cur_node, cur_node->left);
    prune_tree(cur_node, cur_node->right);
}
}
DEBUG_RETURN(0);
}

```

11.3.7 代码的编译和测试

如果你还没有下载本章的源代码，就请下载并将其放到MySQL源代码树根目录下的/sql子目录里。你应该花点儿时间浏览一下那些代码，让自己对那些方法做到心中有数。这样，万一你需要调试这些代码去适应你的系统配置，或者如果你想再添加点儿自己的东西或做练习题，都不至于要临时抱佛脚。把所有的源代码文件都下载回来并浏览之后，还需要把这些文件添加到你的制作文件（Linux平台）或项目文件（Windows平台）里；具体步骤见10.2.4小节里的第5条和第6条。你需要把Attribute和Expression辅助类的源文件添加到你的项目里。把这些文件添加到DBXP项目里以后，编译你的MySQL服务器，编译工作必须不出任何错误地完成。

运行测试

在安装并编译好新代码之后，就可以启动MySQL服务器并运行测试了。可以运行我们在本章开头所创建的测试，也可以再找个MySQL命令行客户端工具来以手动方式输入那些SQL命令。代码清单11-21给出了一个示例测试过程。

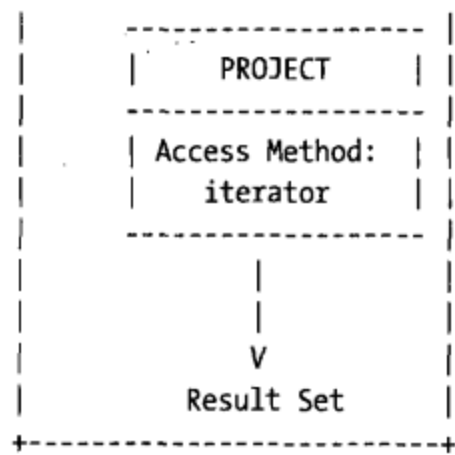
代码清单11-21 测试DBXP查询优化器

```
mysql> SELECT DBXP * FROM staff;
```

```
+-----+
| Execution Path |
+-----+
| expert_mysql.staff |
|   |               |
|   |               |
|   | V             |
|   |               |
|   +-----+       |
|   | PROJECT |       |
|   +-----+       |
|   | Access Method: |
|   | iterator      |
|   +-----+       |
|   |               |
|   | V             |
|   | Result Set    |
|   +-----+       |
+-----+
15 rows in set (0.32 sec)
```

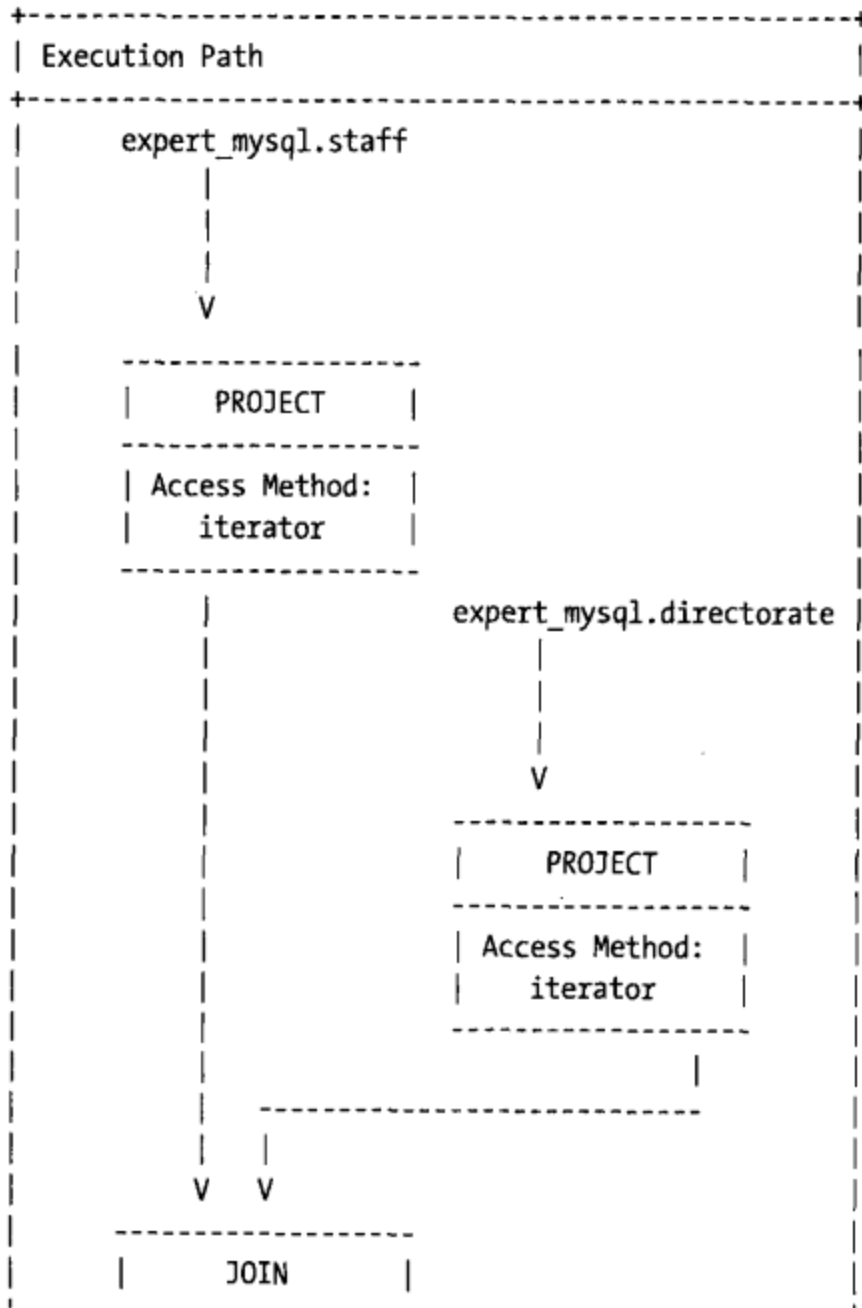
```
mysql> SELECT DBXP id FROM staff WHERE staff.id = '123456789';
```

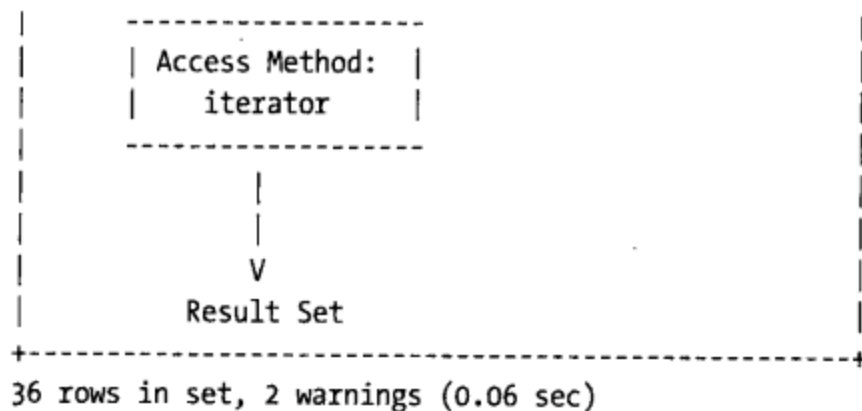
```
+-----+
| Execution Path |
+-----+
| expert_mysql.staff |
|   |               |
|   |               |
|   | V             |
|   |               |
|   +-----+       |
|   | RESTRICT  |       |
|   +-----+       |
|   | Access Method: |
|   | iterator      |
|   +-----+       |
|   |               |
|   | V             |
|   +-----+       |
+-----+
```

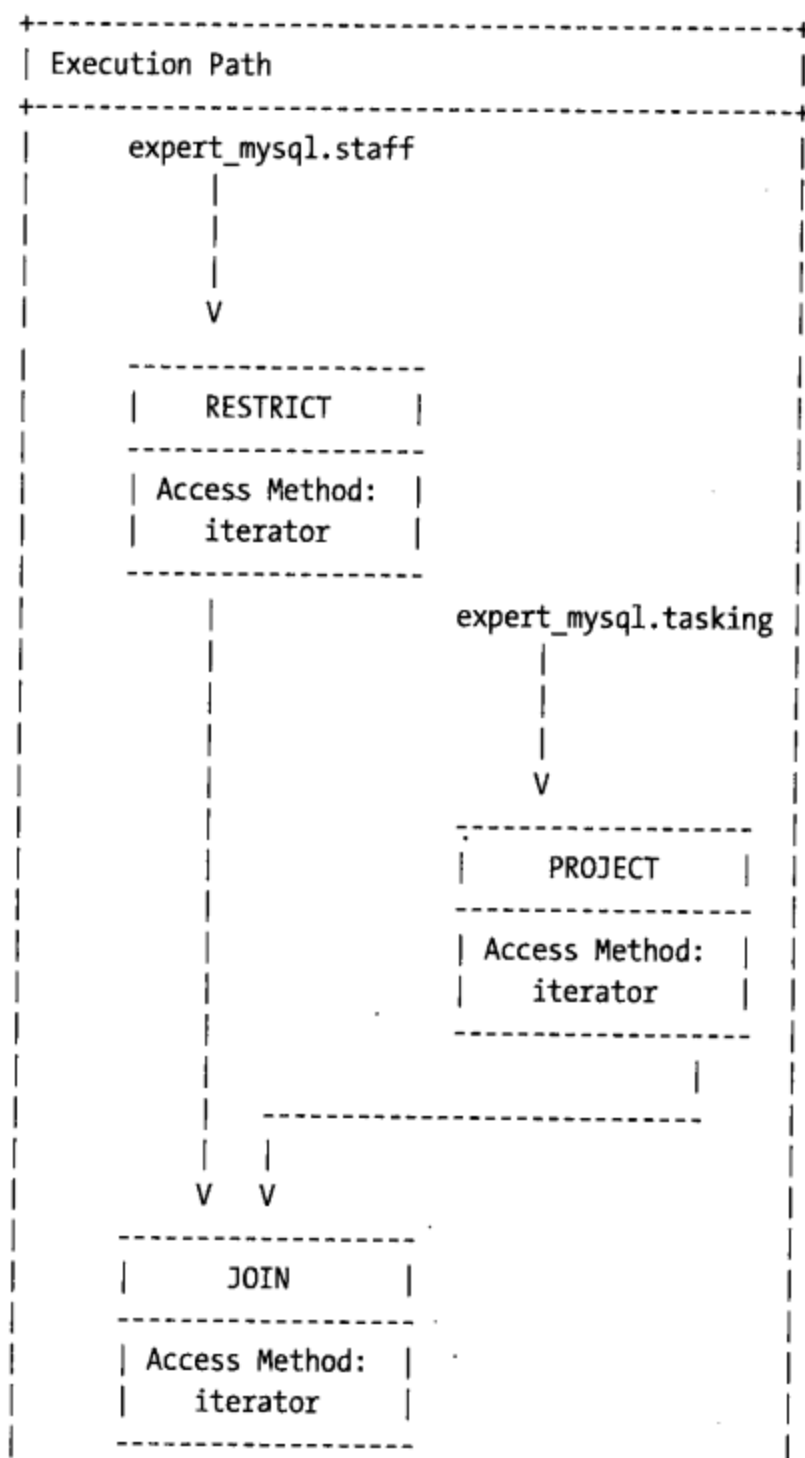
25 rows in set (0.09 sec)

```
mysql> SELECT DBXP id, dir_name FROM staff, directorate
mysql> WHERE staff.dno = directorate.dnumber;
```





```
mysql> SELECT DBXP * FROM staff JOIN tasking ON staff.id = tasking.id
mysql> WHERE staff.id = '123456789';
```



```

      |
      |
      V
Result Set
-----+-----
36 rows in set (0.06 sec)

```

```
mysql>
```

请注意DBXP优化器为各语句生成的执行计划都有哪些不同之处。你可以再试试其他类型的查询语句，看DBXP查询优化器是如何对它们进行优化的。

11.4 小结

本章介绍了最复杂的数据库技术——优化器。文中讲解了如何扩展查询树的概念来引入一个在优化过程中使用树结构的查询优化器。更重要的是，你知道了如何去创建一个启发式查询优化器。启发式优化器的知识可以让你进一步了解DBXP引擎并知道如何使用它来更深入地学习数据库技术，这并不比优化器的内容深。

下一章将对查询执行做更深入的剖析并演示如何实现一种查询树优化策略。DBXP引擎将在下一章最终完成，我将把使用查询树类的启发式查询优化器与一个同样也使用查询树结构的执行过程链接在一起。

练习

下面是几个值得进一步研究的问题，它们在同类问题中很有代表性，你在研究关系数据库技术时，可以把它们作为实验题材（或课堂作业）。

1. 请完成balance_joins()方法的代码。提示：你需要创建一个算法来移动那些在查询树里上下相邻的联结，让限制性最强的联结操作最先执行（位于查询树的最低层）。
2. 请完成cost_optimization()方法的代码。提示：你需要遍历查询树以找出哪些结点可以使用索引。
3. 请仔细阅读DBXP启发式优化器的代码。它能覆盖所有可能的查询情况吗？如果不能，还需要添加哪些规则才能让它做到这一点。
4. 请仔细阅读查询树和DBXP启发式优化器的代码。怎样才能实现查询树类里的DISTINCT结点类型？提示：阅读heuristic_optimization()方法里紧跟在prune_tree()方法后面的代码。
5. 如何修改代码才能让它识别出非法的查询？判断一个查询是否非法的条件是什么？你又该如何测试它们？
6. （高级）MySQL目前还不支持交操作（按照C. J. Date给出的定义）。请修改MySQL解析器，让它能够识别新关键字INTERSECT并处理像SELECT * FROM A INTERSECT B这样的查询。这种操作有什么局限性？处理它们是否需要修改优化器？
7. （高级）如果让你来实现HAVING、GROUP BY和ORDER BY子句，你会怎么做？请修改DBXP优化器使它支持这些子句。

第10章完成的查询树类和在第11章完成的启发式优化器是DBXP查询执行引擎三大组成部分中的前两个。本章将演示如何扩展查询树类去处理投影（project）、限制（restrict）和联结（join）操作，把你带入数据库查询执行的世界。本章首先简要介绍查询执行算法的基本原则，然后开始编写代码。因为有些方法的代码相当冗长，所以本章的代码示例不都是完整的源代码。如果想参照书中的例子做练习，建议你从本书的配套站点下载有关的源代码，而不要从头开始输入这些代码。

12.1 回顾查询执行

查询执行过程就是关系理论中各种操作的实现。这些操作包括投影、限制、联结、叉积（cross-product）、并（union）和交（intersect）。只有少数几种数据库系统实现了并和交操作。

注解 这里所说的并和交操作与SQL语言里的UNION操作符是不一样的。关系理论中的并和交操作属于集合操作，SQL语言里的UNION操作符则是把两个或更多个有着兼容的结果列的SELECT语句结果集简单地拼接在一起。

编写算法来实现这些操作是一件非常简单的事情，所以很多关于关系理论和数据库系统的教材都省略了这部分内容。我认为，对这些算法不能一概而论，联结操作还是很值得讨论的。接下来的几节将对各种关系操作的基本算法进行描述。

12.1.1 投影

投影操作的结果集是原始关系（表）里的属性（列）的一个子集。换句话说，投影操作的结果集所包含的属性要比原始关系少。用户通过在SQL语言的SELECT命令中列出紧跟在SELECT关键字后面的列清单中需要的列表来指定投影。下面这条命令将把first_name和last_name列从staff表投影到结果集里。

```
SELECT first_name, last_name FROM staff
```

用来实现这个操作的投影算法如代码清单12-1所示。

代码清单12-1 投影算法

```

begin
do
  get next tuple from relation
  for each attribute in tuple
    if attribute.name found in column_list
      write attribute data to client
    fi
  while not end_of_relation
end

```

正如你在这份代码清单里看到的那样，这段代码所实现的算法只会把在SELECT命令的列清单里给出的列中的数据发送给客户端。

12.1.2 限制

限制^①操作的结果集是原始关系（表）里的元组（行）的一个子集。因此，限制操作的结果集所包含的元组要比原始关系少。用户通过在SQL语言的SELECT命令中列出紧跟在FROM子句后面的WHERE子句里的条件表达式来指定限制。下面这条命令将把来自staff表的结果集限制在年收入大于65 000.00美元的那些员工。

```
SELECT first_name, last_name FROM staff WHERE salary > 65000.00
```

用来实现这种操作的限制算法如代码清单12-2所示。

代码清单12-2 限制算法

```

begin
do
  get next tuple from relation
  if tuple attribute values match conditions
    write attribute data to client
  fi
  while not end_of_relation
end

```

正如你在这份代码清单里看到的那样，这段代码所实现的算法只会把符合WHERE子句中的条件表达式的元组里的数据发送给客户端。这类算法通常还有一个额外的优化步骤：把条件表达式简化为一个最小集合（比如，消除恒真条件）。

12.1.3 联结

联结操作的结果集包含根据给定条件从两个关系（表）里选取匹配元组（行）。3个或更多个表之间的联结将被分解为 $n-1$ 个联结操作， n 是表的个数。一个涉及3个表（A、B、C）的联结操作将这样来进行：先把两个表联结起来，再把这次联结操作的结果与第3个表联结起来。按照不同的顺序来联结多个表往往会导致不同的中间结果和最终结果。容易通过编程实现的联结顺序有3种：左优先

① 有些书称为“选取”。——译者注

(left-deep)、右优先 (right-deep)、双向优先 (bushy)。这3种顺序用树结构来表示如图12-1所示。

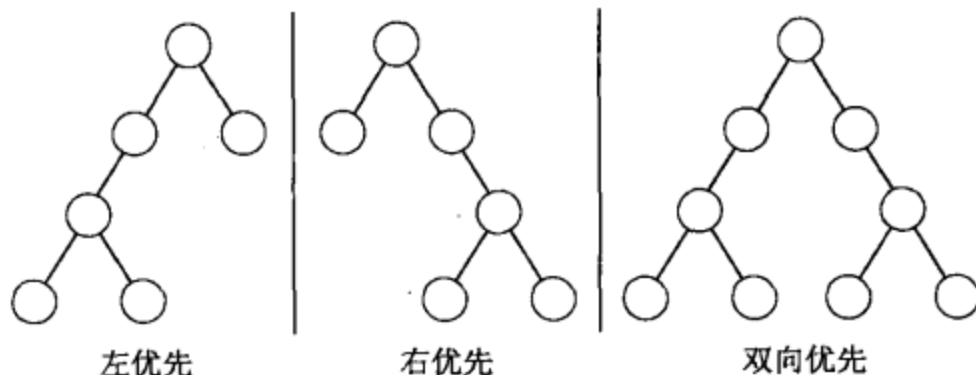


图12-1 用树结构来表示联结顺序

联结操作最常见于“主/细节”关系，即一个表（基表或主表）引用着一个或多个子表（细节表），而在子表中，基表里的每一条记录都匹配细节表里的一条或多条记录。比如说，我们不妨假设你创建了一个customer表来存放顾客的信息，创建了一个orders表来存放顾客订单的信息。customer表是基表，orders表是子表。

```
SELECT customer.name, orders.number
FROM customer JOIN orders on customer.id = orders.customerid
```

用户通过在SQL语言的SELECT命令中列出FROM子句里的表和联结关系来指定联结。比如说，上面这条命令将把来自customer表的记录与orders表里的记录联结起来。请注意，具体到这个例子，联结条件是一个简单的“等于”条件，等式两头是分别来自两个表但含义完全相同的两个列。

用来实现联结操作的算法不像我刚才描述的那几个算法那么简单，这是因为联结操作可以被表示为好几种形式。你可以像上面的例子里那样使用一个简单的表达式如column from table A = column from table B来进行联结，也可以选择其他联结类型来控制输出，让结果集里只包含匹配的行（内联结），只包含来自左、右或两个表的匹配行和不匹配行等。联结操作因而可以细分为内联结（有时也叫作自然联结或等式联结）^①、左外联结、右外联结、全外联结、叉积、并和交。下面几节将对这些操作逐一进行介绍。

注解 有些数据库教材认为叉积、并和交操作是一些离散操作，但我认为它们是联结操作的特殊形式。

1. 内联结

内联结操作的结果集包含一个原始关系（表）里与给定联结条件相匹配的元组（行）的子集。之所以称之为内联结，是因为这种操作的结果集只包含第一个关系中联结条件值与第二个关系里的行的联结条件值匹配的行。

用户通过列出FROM子句里的表和联结关系来在SQL语言的SELECT命令中指定内联结。比如，下面这条命令将把来自staff表的结果集和directorate表联结起来，最终返回的结果集是一份由员工姓名及其所属部门构成的清单，staff表里的行在directorate表里没有与之匹配的行时（即staff表里dept_id字段是空值null的那名员工），不会出现在这份清单里。

```
SELECT staff.last_name, staff.dept_name
FROM staff JOIN directorate on staff.dept_id = directorate.id
```

^① 自然联结是等式联结的一种特殊形式，从后者的结果集里剔除重复的属性就得到了前者的结果集。

注解 在绝大多数数据库系统里，内联结是默认的联结操作，所以关键字 INNER 通常是可选的。

用来实现这种操作的内联结算法如代码清单12-3所示。这个算法只是众多联结算法当中的一种。这个算法是合并型联结（merge-join）算法的一种变体。我们完全可以选用另外一种与之等效的算法（比如某种嵌套循环型联结算法）来实现这一操作。

代码清单12-3 联结算法

```
begin
  sort relation a as rel_a on join column(s)
  sort relation b as rel_b on join column(s)
do
  get next tuple from rel_a
  get next tuple from rel_b
  if join column values match join conditions
    write attribute data to client
  fi
  check rewind conditions
while not end_of_rel_a and not end_of_rel_b
end
```

绝大多数数据库系统还允许用户把联结条件写在 WHERE 子句里，如下所示。有些数据库专家反对这种做法，认为这有可能导致数据库系统把它误认为是一条普通的 SELECT 命令。不过，因为用起来很方便，在功能上也没什么差异，所以绝大多数数据库系统都允许这样做，它们的优化器也能正确地理解这种“错误”。

```
SELECT staff.last_name, directorate.dept_name
FROM staff, directorate WHERE staff.dept_id = directorate.id
```

正如你在代码清单12-3里看到的那样，实现这个算法的代码要求使用排序，排序操作需要在联结列上对表中的各行进行排序，以便这个算法可以查明所有的匹配都应当是各行之间任意成对的条件值。为了说明这一点，代码清单12-4里给出了两个表作为例子。

代码清单12-4 示例联结操作所使用的表（未排序）

staff table

first_name	last_name	id	dept_id
Bill	Smith	123456789	5
Aaron	Hill	987987987	4
Alicia	Wallace	330506781	4
Howard	Bell	333445555	5
William	Wallace	220059009	<null>
Steven	Marrow	401550022	5
Tamra	English	453453453	5
Chad	Borg	990441234	1
Lillian	Wallace	987654321	4

directorale table

id	dept_name
5	Research
4	Administration
6	Marketing
1	Headquarters

注解 有些数据库系统（比如MySQL）返回的是一些未经过排序的行。为了强调，所显示的例子都是按照内部排序的顺序给出的。

请注意，这两个表都没有经过排序。如果在执行联结操作之前不对表进行排序，你需要读取一个表里的每一行与第二个表里的所有行进行比较。比如，如果按照所给顺序读取staff表，你要从directorale表为第一个联结读取一行，为staff表里的下一行从directorale表读出两行，再往后需要你读出2、1、1、1、4、2行。于是，为了完成这次联结操作，总共需要对directorale表进行14次读操作。如果把这两个表排序成代码清单12-5的样子，就可以避免对directorale表进行这么多次读操作。

代码清单12-5 示例联结操作所使用的表（已按联结列排序）

staff table

first_name	last_name	id	dept_id
William	Wallace	220059009	<null>
Chad	Borg	990441234	1
Aaron	Hill	987987987	4
Alicia	Wallace	330506781	4
Lillian	Wallace	987654321	4
Howard	Bell	333445555	5
Steven	Marrow	401550022	5
Tamra	English	453453453	5
Bill	Smith	123456789	5

directorale table

id	dept_name
1	Headquarters
4	Administration
5	Research
6	Marketing

但这又引出了另一个问题。你凭什么判断不需要对这两个表再进行一次读操作呢？请注意代码清单12-3里的内联结算法的最后一步，它是在实现这个算法时需要特别注意的一个地方。在这里，你必须能够重复使用一个已经被读过的行才能把第一个表里某个行与第二个里的众多行进行比较。如

果必须在两个表里前移或后移一行的话，这件事情就有点棘手了。

如果你以手动方式使用排过序的staff和directorate表（把staff表视为算法里的rel_a关系）来进行这次联结操作，将看到这个算法需要重复使用directorate表里id是4的那个行2次，需要重复使用id是5的那个行3次。这种重复使用某个行的情况有时被称为“回卷”（rewind）表的读指针。这个例子的结果集如代码清单12-6所示。

代码清单12-6 内联结操作的结果集示例

last_name	dept_name
Borg	Headquarters
Hill	Administration
Wallace	Administration
Wallace	Administration
Bell	Research
Marrow	Research
English	Research
Smith	Research

2. 外联结

外联结与内联结在原理上是一样的，但在外联结操作里，我们想要获得的是来自左、右，或两个表的所有行。换句话说，不管有没有在另一个表里找到与之匹配的行，我们都想看到来自某个特定的表（左表、右表或两个表）的所有行。这些操作大同小异，我们可以用同一种外联结算法的变体把它们分别实现出来。

在SQL语言中的SELECT命令里，用户在FROM子句里列出所需的条件并触发其中的一个选项（left、right或full）来指定外联结。如果用户没有给出这个选项，大部分数据库会使用left作为它的默认值。比如说，下面这条命令将把来自staff表和来自directorate表里的结果集联结起来，最终返回的结果集是一份由全体员工姓名及其所属部门构成的清单，在directorate表里没有与之匹配的行的staff表里的行——staff表里dept_id字段是空值null的那名员工——也会出现在这份清单里。

```
SELECT staff.last_name, directorate.dept_name
FROM staff LEFT OUTER JOIN directorate on staff.dept_id = directorate.id
```

请注意这与内联结操作的区别，来自左表的行都将出现在结果集里。代码清单12-7给出了一个基本的外联结算法。我将在本章稍后的有关小节里利用这个算法来实现左、右、全3种外联结操作。

代码清单12-7 外联结算法

```
begin
  sort relation a as rel_a on join column(s)
  sort relation b as rel_b on join column(s)
do
  get next tuple from rel_a
  get next tuple from rel_b
  if type is FULL
    if join column values match join conditions
```

```

        write attribute data from both tuples to client
    else
        if rel_a has data
            write NULLS for rel_b
        else if rel_b has data
            write NULLS for rel_a
        fi
    else if type is LEFT
        if join column values match join conditions
            write attribute data from rel_a to client
        else
            if rel_a has data
                write NULLS for rel_a
            fi
        else if type is RIGHT
            if join column values match join conditions
                write attribute data from rel_b to client
            else
                if rel_b has data
                    write NULLS for rel_a
                fi
            fi
        check rewind conditions
    while not end_of_rel_a and not end_of_rel_b
end

```

接下来，看看各种外联结操作的例子。

● 左外联结

左外联结（left outer join）操作的结果集包含着左表里的所有行，如果左表里的某个行在右表里没能找到与联结条件相匹配的行，来自右表的列将被返回为空值null。

```

SELECT staff.last_name, directorate.dept_name
FROM staff LEFT OUTER JOIN directorate on staff.dept_id = directorate.id

```

代码清单12-8给出了用上面这条命令对示例表进行左外联结操作的结果集。

代码清单12-8 左外联结操作的结果集示例

last_name	dept_name
Wallace	<null>
Borg	Headquarters
Hill	Administration
Wallace	Administration
Wallace	Administration
Bell	Research
Marrow	Research
English	Research
Smith	Research

● 右外联结

右外联结 (right outer join) 操作的结果集包含着右表里的所有行, 如果右表里的某个行在左表里没能找到与联结条件相匹配的行, 来自左表的列将被返回为空值null。

```
SELECT staff.last_name, directorate.dept_name
FROM staff RIGHT OUTER JOIN directorate on staff.dept_id = directorate.id
```

代码清单12-9给出了用上面这条命令对样板表进行右外联结操作的结果集。

代码清单12-9 右外联结操作的结果集示例

last_name	dept_name
Borg	Headquarters
Hill	Administration
Wallace	Administration
Wallace	Administration
Smith	Research
Bell	Research
Marrow	Research
English	Research
<null>	Marketing

● 全外联结

全外联结 (full outer join) 操作的结果集包含着两个表里的所有行, 如果其中一个表里的某个行在另一个表里没能找到与联结条件相匹配的行, 来自另一个表的列将被返回为空值null。

```
SELECT staff.last_name, directorate.dept_name
FROM staff FULL OUTER JOIN directorate on staff.dept_id = directorate.id
```

代码清单12-10给出了用上面这条命令对样板表进行全外联结操作的结果集。

代码清单12-10 全外联结操作的结果集示例

last_name	dept_name
Wallace	<null>
Borg	Headquarters
Hill	Administration
Wallace	Administration
Wallace	Administration
Bell	Research
Marrow	Research
English	Research
Smith	Research
<null>	Marketing

3. 叉积

叉积 (cross-product) 操作的结果集是左、右两个表里的行的全排列组合。如果左表包含着n个行, 右表包含着m个行, 它们的叉积结果集将包含着n×m个行。虽然在概念上很简单, 但并非所有的数据

库系统都支持叉积操作。

注解 有些数据库系统把SELECT * FROM table, table2这样的查询命令解释为一个叉积查询。此时，因为没有任何联结条件，所以表table1里的每一个行与表table2里的每一个行都可以匹配上。如果你在MySQL上试一下这条命令，将会看到MySQL正是按照这种方式来支持叉积操作的。

在SQL语言中的SELECT命令里，把FROM子句里的JOIN关键字替换为CROSS，就可以得到一条叉积查询命令。你也许会认为这种操作没有什么实际用途，但它的用途却比你想象得大得多。比如说，你正在模拟一个人工智能项目，用了一个表来存放机器人可能采取的下一个动作，用了另一个表来存放各种试验条件。如果你想知道机器人在各种试验条件下都有可能采取哪些动作，就需要对那两个表进行叉积操作，以获得一个包含着所有可能性组合的结果集。代码清单12-11给出了一个这样的例子。

代码清单12-11 叉积的应用场景示例

```
CREATE TABLE next_stim
SELECT source, stimuli_id FROM stimuli WHERE likelihood >= 0.75
```

source	stimuli_id
obstacle	13
other_bot	14
projectile	15
chasm	23

```
CREATE TABLE next_moves
SELECT move_name, next_move_id, likelihood FROM moves WHERE likelihood >= 0.90
```

move_name	next_move_id	likelihood
turn left	21	0.25
reverse	18	0.40
turn right	22	0.45

```
SELECT * FROM next_stim CROSS next_moves
```

source	stimuli_id	move_name	next_move_id	likelihood
obstacle	13	turn left	21	0.25
obstacle	13	reverse	18	0.40
obstacle	13	turn right	22	0.45
other_bot	14	turn left	21	0.25
other_bot	14	reverse	18	0.40
other_bot	14	turn right	22	0.45
projectile	15	turn left	21	0.25
projectile	15	reverse	18	0.40
projectile	15	turn right	22	0.45
chasm	23	turn left	21	0.25

chasm	23	reverse	18	0.40	
chasm	23	turn right	22	0.45	
+-----+-----+-----+-----+-----+					

代码清单12-12给出了一种叉积算法。请注意，这个算法是分两步完成操作的：先把行组合在一起，再剔除那些雷同的行。

代码清单12-12 叉积算法

```
begin
do
  get next tuple from rel_a
do
  get next tuple from rel_b
  write tuple from rel_a concat tuple from rel_b to client
while not end_of_rel_b
while not end_of_rel_a
remove duplicates from temp_table
return data from temp_table to client
end
```

正如你在这份代码清单里看到的那样，这段代码使用了一个两层嵌套循环：内循环负责把左、右两个表里的行组合在一起，外循环负责剔除那些重复雷同的行。这是一个嵌套循环型算法。

4. 并

这里所说的并操作是一种集合操作：把两个表里的所有行合在一起并剔除那些雷同的行。这与SQL语言中的并（UNION）操作是不一样的：SQL语言中的并操作只是把多个SELECT命令的结果集（它们必须有同样的列清单）首尾相接地合并在一起，不剔除那些雷同的行。与其他类型的联结操作不同，并操作通常被实现为两步：先合并表，再剔除那些重复雷同的行。

在SQL语言中的SELECT命令里，把FROM子句里的JOIN关键字替换为UNION，就可以得到一条并查询命令。我们不妨假设你需要用两个员工表（一个是美国分公司的，一个是加拿大分公司的）生成一份名单并保证所有的员工都必须在这份名单（结果集）里不多不少地出现一次。你会使用下面这条命令来完成这个并操作。

```
SELECT * from us_employees UNION ca_employees
```

我们来仔细分析一下这个例子。代码清单12-13给出了美国分公司和加拿大分公司的员工表。稍微留意一下就会发现，有两名员工的名字出现在了两个表里。如果你使用的是SQL语言中的UNION命令，这两名员工将被统计两次。代码清单12-14给出了对这两个表进行并操作的正确结果。

代码清单12-13 示例Employee表

US employees table			
first_name	last_name	id	dept_id
+-----+-----+-----+-----+			
Chad	Borg	990441234	1
Alicia	Wallace	330506781	4
Howard	Bell	333445555	5
Tamra	English	453453453	5

Bill	Smith	123456789	5
------	-------	-----------	---

Canada employees table

first_name	last_name	id	dept_id
William	Wallace	220059009	<null>
Aaron	Hill	987987987	4
Lillian	Wallace	987654321	4
Howard	Bell	333445555	5
Bill	Smith	123456789	5

代码清单12-14 并的结果集示例

first_name	last_name	id	dept_id
Chad	Borg	990441234	1
Alicia	Wallace	330506781	4
Howard	Bell	333445555	5
Tamra	English	453453453	5
Bill	Smith	123456789	5
William	Wallace	220059009	<null>
Aaron	Hill	987987987	4
Lillian	Wallace	987654321	4

代码清单12-5给出了这个例子所使用的并算法。请注意，这个算法是分两个步骤完成操作的：先合并表，再剔除那些重复雷同的行。

代码清单12-15 并算法

```

begin
do
  get next tuple from rel_a
  write tuple from rel_a to temp_table
  get next tuple from rel_b
  write tuple from rel_b to temp_table
  while not end_of_rel_a or end_of_rel_b
  remove duplicates from temp_table
  return data from temp_table to client
end

```

5. 交

这里所说的交操作是一种集合操作：把两个表里重复雷同的行都找出来并剔除那些重复的行。当然，要想执行这个操作，那两个表必须有着同样的构造。

在SQL语言中的SELECT命令里，把FROM子句里的JOIN关键字替换为INTERSECT就可以得到一条交查询命令。我们不妨假设你需要用两个员工表（一个是美国分公司的，一个是加拿大分公司的）生成一份名单并保证所有“公共”员工都必须在这份名单（结果集）里不多不少地出现一次。你会使用下面

这条命令来完成这个并操作。

```
SELECT * from us_employees INTERSECT ca_employees
```

我们来仔细分析一下这个例子。代码清单12-13给出了美国分公司和加拿大分公司的员工表。稍微留意一下就会发现，有两名员工的名字出现在了两个表里。代码清单12-16给出了对这两个表进行交操作的正确结果。代码清单12-17给出了这个例子所使用的交算法。

代码清单12-16 交操作的结果集示例

first_name	last_name	id	dept_id
Howard	Bell	333445555	5
Bill	Smith	123456789	5

代码清单12-17 交算法

```
begin
do
  get next tuple from rel_a
  get next tuple from rel_b
  if join column values match intersection conditions
    write tuple from rel_a to client
  while not end_of_rel_a or end_of_rel_b
end
```

12.2 DBXP 查询执行

DBXP查询执行引擎是利用经过优化的查询树实现的，这个树结构本身控制着查询执行的全过程。在执行一个查询的时候，根结点会为自己的每一个子结点调用一次get_next()方法，根结点的子结点又会为它们自己的每一个子结点调用一次get_next()方法，这个过程将一直持续到那些只包含着单个表的引用指针的树结点（叶结点）为止。请看下面这条查询命令。

```
SELECT col1, col2 FROM table_a JOIN
(SELECT col2, col8 FROM table_b WHERE col6 = 7)
ON col8 WHERE table_a.col7 > 14
```

这个查询将对表table_a和表table_b的一个子集进行联结操作。请注意，我使用了一个子查询来生成那个子集。子查询机制很容易用查询树来实现，只要把子查询表示为一个子树就可以了。图12-2给出了这个概念的示意图。

查询树各个结点上的操作是以每次处理一个行的方式进行的。如果某个结点上的操作会产生结果，该结点会在结果产生后把它立刻传递到查询树中的下一个操作（它的父结点）。如果某个结点上的操作不产生结果，控制权将留在这个结点直到产生一个结果为止。随着操作的进行，叶结点产生的结果将经过每一个父结点到达根结点。在根结点上的操作完成之后，结果元组将被发送给客户端。这样一来，查询执行

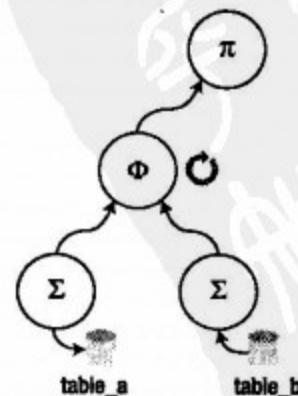


图12-2 用查询树来执行查询命令

就会更快地产生结果，因为比起为整个操作集执行查询后再把结果一次性发送给客户端的做法，这种策略会让客户端更快地看到数据。

Query_tree类能够把执行查询所需要的所有操作——包括投影、限制和联结操作在内——容纳在自己的内部。从查询优化阶段进入查询执行阶段的标志是调用prepare()方法。prepare()方法将遍历整个查询树，对所有的结点进行查询执行前的初始化。查询的执行是通过一个while循环实现的，这个循环将从根结点开始，向查询树发出一系列对get_next()方法的调用，直到结果集里的元组全都被返回给客户端为止。脉冲是对get_next()方法的一次调用，它将向下波及整个查询树。每个结点将从它的左子结点开始，把这个脉冲依次传递给它的每一个子结点。DBXP执行引擎里的限制、投影和联结操作被分别实现为带参数的do_restrict()、do_project()和do_join()方法^①。这些方法的输入参数是一条或两条元组，返回值是空值(null)或一条元组。空值(null)的含义是通过输入参数传递来的元组不满足当前操作。比如说，do_restrict()方法需要一条元组作为输入参数，它将使用expression类对这条元组里的值进行求值；如果求值结果是false，它将返回空值null。如果求值结果是true，则返回相同的元组^②。

这个过程将在查询树中的每个结点上演，直到一条元组被传递到根结点为止。根结点产生的元组记录将由外层while循环（这个外层while循环包含着上面提到的那个while循环）再做一些修饰性的处理，然后借用现成的MySQL客户端通信协议被发送给客户端。这种查询执行方式被称为流水线方式，因为结点遍历的方式是沿着各个结点遍历整棵树，从而执行查询中的各种操作。

12.2.1 测试的设计

为一个查询执行引擎创建一套完备的测试需要编写大量的SQL代码，才能覆盖所有可能的执行路径。简单地说，你将需要创建一个测试去测试有可能出现的一切查询，合法的和不合法的都要覆盖到。DBXP查询执行引擎还远称不上完备——投影和限制操作的实现是完备的，但do_join()方法只实现了内联结操作。我希望你能在自己的机器上——在把DBXP引擎调试稳定之后——把本章没有实现的联结操作都实现出来。

怀着这个想法，我们现在需要设计几个基本的查询命令去测试DBXP执行引擎将如何处理查询。代码清单12-18给出了一个可以用来测试DBXP查询引擎的示例测试文件。你可以添加自己的查询命令来测试DBXP引擎还有哪些需要改进的地方。

代码清单12-18 DBXP查询执行引擎的示例测试文件（ExpertMySQLCh12.test）

```
#
# Sample test to test the SELECT DBXP execution
#

# Test 1:
SELECT DBXP first_name, last_name, sex, id FROM staff;

# Test 2:
```

① 集合操作（并和交）被实现为联结操作的特例。

② 事实上，所有的记录都是通过相应的引用指针来传递的，返回值将通过同一个引用指针被返回。


```

SELECT DBXP id FROM staff;

# Test 3:
SELECT DBXP dir_name FROM directorate;

# Test 4:
SELECT DBXP id, dir_name FROM staff
JOIN directorate ON staff.mgr_id = directorate.dir_head_id;

# Test 5:
SELECT DBXP * FROM staff WHERE staff.id = '123456789';

# Test 6:
SELECT DBXP first_name, last_name FROM staff JOIN directorate
WHERE staff.mgr_id = directorate.dir_head_id and directorate.dir_code = 'N41';

# Test 7:
SELECT DBXP * FROM directorate
JOIN building ON directorate.dir_code = building.dir_code;

# Test 8:
SELECT DBXP directorate.dir_code, dir_name, building, dir_head_id
FROM directorate JOIN building ON directorate.dir_code = building.dir_code;

```

提示 这些例子使用的数据库收录在本书的附录里。

如果你打算使用MySQL Test Suite工具来创建和运行代码清单12-18里的测试,请参考本书第4章给出的步骤来进行。

12.2.2 更新 SELECT DBXP 命令

现在,因为我们已经有了一种可以用来执行查询的手段,就可以把DBXP_select_command()方法里的那些“假”代码替换为真正的SELECT命令处理代码了。这个方法将完成以下工作:检查表上的访问权限,打开并锁定表,执行查询,把结果发送给客户端,解除表上的锁定。代码清单12-19给出了最终完成的DBXP_select_command()方法。

代码清单12-19 最终完成的SELECT DBXP命令

```

/*
  Perform Select Command

  SYNOPSIS
    DBXP_select_command()
    THD *thd          IN the current thread

  DESCRIPTION
    This method executes the SELECT command using the query tree and optimizer.

  RETURN VALUE

```

```

    Success = 0
    Failed = 1
*/
int DBXP_select_command(THD *thd)
{
    bool res;
    READ_RECORD *record;
    select_result *result = thd->lex->result;

    DEBUG_ENTER("DBXP_select_command");

    /* Prepare the tables (check access, locks) */
    res = check_table_access(thd, SELECT_ACL, thd->lex->query_tables, 0);
    if (res)
        DEBUG_RETURN(1);
    res = open_and_lock_tables(thd, thd->lex->query_tables);
    if (res)
        DEBUG_RETURN(1);
    /* Create the query tree and optimize it */
    Query_tree *qt = build_query_tree(thd, thd->lex,
        (TABLE_LIST*) thd->lex->select_lex.table_list.first);
    qt->heuristic_optimization();
    qt->cost_optimization();
    qt->prepare(qt->root);
    if (!(result= new select_send()))
        DEBUG_RETURN(1);

    /* use the protocol class to communicate to client */
    Protocol *protocol= thd->protocol;

    /* write the field list for returning the query results */
    if (protocol->send_fields(&qt->result_fields,
        Protocol::SEND_NUM_ROWS | Protocol::SEND_EOF))
        DEBUG_RETURN(1);

    /* pulse the execution engine to get a row from the result set */
    while (!qt->Eof(qt->root))
    {
        record = qt->get_next(qt->root);
        if (record != NULL)

            /* send the data to the client */
            send_data(protocol, qt->result_fields, thd);
    }
    send_eof(thd);

    /* unlock tables and cleanup memory */
    qt->cleanup(qt->root);
    mysql_unlock_read_tables(thd, thd->lock);

```

```

delete qt;
DEBUG_RETURN(0);
}

```

这个实现已经有了执行查询命令所需要的所有元素。它首先会去检查表上的访问权限并尝试打开表。如果这些步骤都成功完成的话，DBXP查询引擎就将正式登场。DBXP查询引擎将先创建查询树，再对查询树进行优化，最后在一个循环里执行查询。请注意，那是一个典型的“不到文件尾不停”的while循环，它会在根结点上反复调用get_next()方法，如果从该方法返回了一条元组（记录），有关代码将把它输出给客户端。否则，它将继续调用get_next()方法直到遇到eof（end-of-file）标志为止。在把所有的元组都处理完之后，有关代码将释放所有被占用的内存并解除表上的锁定。我把负责向客户端发送数据的代码放在了那些查询树方法以外的一个地方，这多少可以简化为各种关系操作编写代码实现的工作。正如你将在接下来的几小节看到的那样，我实现的查询树方法与相应的理论算法非常相似。

12.2.3 DBXP 算法

现在，让DBXP查询引擎自身运转起来的代码已经完成了。让我们把注意力转到DBXP项目的Query_tree类是如何实现各种关系操作上来。

1. 投影

在DBXP查询引擎里，投影操作是在Query_tree类里的一个名为do_project()的方法里实现的。这个方法很容易实现，因为MySQL基类为我们提供了一个可以快速完成投影操作的办法。我们用不着编写一个循环去遍历那些行里的属性，可以使用MySQL基类把数据发送给客户端。

do_project()方法可以被简化为只做两件事：一是把当前行保存到缓冲区，二是把行返回给查询树里的下一个结点。当控制权回到DBXP_select_command()方法手里的时候，一个名为send_data()的辅助方法将把数据发送给客户端。代码清单12-20给出了do_project()方法的源代码。

代码清单12-20 DBXP投影方法

```

/*
Perform project operation.

SYNOPSIS
do_project()
query_node *qn IN the operational node in the query tree.
READ_RECORD *t -- the tuple to apply the operation to.

DESCRIPTION
This method performs the relational model operation entitled
"project". This operation is a narrowing of the result set
vertically by restricting the set of attributes in the
output tuple.

NOTES
Returns 0 (null) if no tuple satisfies child operation
(does NOT indicate the end of the file or end of query
operation. Use Eof() to verify.

```

```

RETURN VALUE
    Success = new tuple with correct attributes
    Failed = NULL
*/
READ_RECORD *Query_tree::do_project(query_node *qn, READ_RECORD *t)
{
    DEBUG_ENTER("do_project");
    if (t != NULL)
    {
        if (qn == root)
        {
            /*
             * If the left table isn't NULL, copy the record buffer from
             * the table into the record buffer of the relations class.
             * This completes the read from the storage engine and now
             * provides the data for the projection which is accomplished
             * in send_data().
             */
            if (qn->relations[0] != NULL)
                memcpy((byte *)qn->relations[0]->table->record[0],
                    (byte *)t->rec_buf,
                    qn->relations[0]->table->s->rec_buff_length);
        }
        DEBUG_RETURN(t);
    }
}

```

请注意，这个方法必须完成的工作只有一个：把从存储引擎那里读来的数据复制到表对象的记录缓冲区里。我是这样完成这一工作的：把从存储引擎那里读来的READ_RECORD变量的内存复制到表的第一个READ_RECORD缓冲区里去，需要复制的字节个数由表的rec_buff_length属性决定。

2. 限制

在DBXP查询引擎里，限制操作是在Query_tree类里的一个名为do_restrict()的方法里实现的。这段代码使用了Query_tree类的where_expr成员变量，它包含着Expression辅助类的一个实例。因此，限制操作可以简单地通过调用Expression类的evaluate()方法来实现。代码清单12-21给出了do_restrict()方法的源代码。

代码清单12-21 DBXP限制方法

```

/*
Perform restrict operation.

SYNOPSIS
do_restrict()
query_node *qn IN the operational node in the query tree.
READ_RECORD *t -- the tuple to apply the operation to.

DESCRIPTION
This method performs the relational model operation entitled
"restrict". This operation is a narrowing of the result set
horizontally by satisfying the expressions listed in the
where clause of the SQL statement being executed.

```



```

RETURN VALUE
    Success = true
    Failed = false
*/
bool Query_tree::do_restrict(query_node *qn, READ_RECORD *t)
{
    bool found = false;

    DEBUG_ENTER("do_restrict");
    if (qn != NULL)
    {
        /*
            If the left table isn't NULL, copy the record buffer from
            the table into the record buffer of the relations class.
            This completes the read from the storage engine and now
            provides the data for the projection which is accomplished
            in send_data().

            Lastly, evaluate the where clause. If the where clause
            evaluates to true, we keep the record else we discard it.
        */
        if (qn->relations[0] != NULL)
            memcpy((byte *)qn->relations[0]->table->record[0],
                (byte *)t->rec_buf,
                qn->relations[0]->table->s->rec_buff_length);
        if (qn->where_expr != NULL)
            found = qn->where_expr->evaluate(qn->relations[0]->table);
    }
    DEBUG_RETURN(found);
}

```

在找到一个匹配的时候，数据将被复制到表的记录缓冲区。这将把当前记录缓冲区里的数据与表对应起来。不仅如此，这还使得我们可以利用众多的MySQL方法来处理各有关字段，并把数据发送给客户端。

3. 联结

在DBXP查询引擎里，联结操作是在Query_tree类里的一个名为do_join()的方法里实现的。这段代码使用了Query_tree类的join_expr成员变量，该成员变量包含着Expression辅助类的一个实例。因此，联结操作可以简化成通过调用Expression类的evaluate()方法来实现。

这个方法是所有DBXP代码当中最复杂的，这在很大程度上是因为联结操作可以细分为许多不同的类型。前面内容里的理论算法和示例已经揭示了这一点。为了帮助大家看懂do_join()方法的源代码，我将在这里多说几句。代码清单12-22是do_join()方法经过简化的伪代码。

代码清单12-22 DBXP联结算法

```

begin
    if preempt_pipeline
        do
            if no left child

```

```

        get next tuple from left relation
    else
        get next tuple from left child
    fi
    insert tuple in left buffer in order by join column for left relation
until eof
do
    if no right child
        get next tuple from right relation
    else
        get next tuple from right child
    fi
    insert tuple in right buffer in order by join column for right relation
until eof
fi
if left record pointer is NULL
    get next tuple from left buffer
fi
if right record pointer is NULL
    get next tuple from right buffer
fi
if there are tuples to process
    write attribute data of both tuples to table record buffers
    if join column values match join conditions
        check rewind conditions
        clear record pointers
        check for end of file
        set return record to left record pointer (indicates a match)
    else if left join value < right tuple join value
        set return record to NULL (no match)
        set left record pointer to NULL
    else if left join value > right tuple join value
        set return record to NULL (no match)
        set right record pointer to NULL
    fi
else
    set return record to NULL (no match)
fi
end

```

因为do_join()的方法需要在get_next()方法里被反复调用，所以我对算法做了些修改以便能够使用query_node对象的preemp_pipeline成员变量。这个变量会在查询树的执行过程正式开始之前由prepare()方法设置为TRUE。do_join()方法将根据这个变量来判断这是不是自己的第一次调用；如果是，它将先去创建一些必要的临时缓冲区，而查询树的遍历活动将暂停下来直到联结操作找到第一个匹配（或到达文件尾）为止。

这个算法使用了两个缓冲区来保存从表（联结操作涉及两个表）里读出来并经过排序的行。这两个缓冲区用来保存联结操作的中间结果，我使用了两个记录指针来分别代表这两个缓冲区。如果找到了一个匹配，这两个指针都将被设置为NULL，这将强制有关代码去读取下一条记录。如果对联结条件

求值的结果表明左表里的联结值少于右表，左记录指针将被设置为NULL；这将使得do_join()方法在下次调用时从左记录缓冲区读取一条新记录。类似的，如果左表里的联结值多于右表，右记录指针将被设置为NULL，do_join()方法在下次调用时将从右记录缓冲区读取一条新记录。

对do_join()方法的基本情况就解释到这里，接下来请看它的源代码。代码清单12-23给出了do_join()方法的源代码。

注解 我没有使用辅助函数来为联结操作的第一步创建临时缓冲区，因为这样我才可以把代码集中在一个地方以便于调试。我纯粹是出于个人理由才这样做的。如果你想少打些字并提高代码的可读性，完全可以把这部分代码编写成一个辅助函数。

代码清单12-23 DBXP联结方法

```

/*
Perform join operation.

SYNOPSIS
do_join()
query_node *qn IN the operational node in the query tree.
READ_RECORD *t -- the tuple to apply the operation to.

DESCRIPTION
This method performs the relational model operation entitled
"join". This operation is the combination of two relations to
form a composite view. This algorithm implements ALL variants
of the join operation.

NOTES
Returns 0 (null) if no tuple satisfies child operation (does
NOT indicate the end of the file or end of query operation.
Use Eof() to verify.

RETURN VALUE
Success = new tuple with correct attributes
Failed = NULL
*/
READ_RECORD *Query_tree::do_join(query_node *qn)
{
    READ_RECORD *next_tup;
    int i;
    TABLE *ltable = NULL;
    TABLE *rtable = NULL;
    Field *fright = NULL;
    Field *fleft = NULL;
    record_buff *lprev;
    record_buff *rprev;
    expr_node *expr;

    DBUG_ENTER("do_join");

```

```

if (qn == NULL)
    DEBUG_RETURN(NULL);

/* check join type because some joins require other processing */
switch (qn->join_type)
{
    case (jnINNER) :
    case (jnLEFTOUTER) :
    case (jnRIGHTOUTER) :
    case (jnFULLOUTER) :
    {

        /*
         * preempt_pipeline == true means we need to stop the pipeline
         * and sort the incoming rows. We do that by making an in-memory
         * copy of the record buffers stored in left_record_buff and
         * right_record_buff
         */
        if (qn->preempt_pipeline)
        {
            left_record_buff = NULL;
            right_record_buff = NULL;
            next_tup = NULL;

            /* Build buffer for tuples from left child. */
            do
            {
                /* if left child exists, get row from it */
                if (qn->left != NULL)
                    lbuff = get_next(qn->left);

                /* else, read the row from the table (the storage handler */
                else
                {
                    /*
                     * Create space for the record buffer and
                     * store pointer in lbuff
                     */
                    lbuff = (READ_RECORD *) my_malloc(sizeof(READ_RECORD),
                                                         MYF(MY_ZEROFILL | MY_WME));
                    lbuff->rec_buf =
                        (byte *) my_malloc(qn->relations[0]->table->s->rec_buff_length,
                                                         MYF(MY_ZEROFILL | MY_WME));

                    /* check for end of file. Store result in eof array */
                    qn->eof[0] =
                        qn->relations[0]->table->file->rnd_next(lbuff->rec_buf);
                    if (qn->eof[0] != HA_ERR_END_OF_FILE)
                        qn->eof[0] = false;
                    else

```



```

    {
        lbuff = NULL;
        qn->eof[0] = true;
    }
}
/* if the left buffer is not null, get a new row from table */
if (lbuff != NULL)
{
    /* we need the table information for processing fields */
    if (qn->left == NULL)
        ltable = qn->relations[0]->table;
    else
        ltable = get_table(qn->left);
    if (ltable != NULL)
        memcpy((byte *)ltable->record[0], (byte *)lbuff->rec_buf,
            ltable->s->rec_buff_length);

    /* get the join expression */
    expr = qn->join_expr->get_expression(0);
    Field *cur_field = (Field *)expr->left_op;
    for (Field **field = ltable->field; *field; field++)
        if (strcasecmp((*field)->field_name, cur_field->field_name)==0)
            fleft = (*field);

    /*
        If field was found, add the row to the in-memory buffer
        ordered by the join column.
    */
    if ((fleft != NULL) && (!fleft->is_null()))
        insertion_sort(true, fleft, lbuff);
}
} while (lbuff != NULL);
/* Build buffer for tuples from right child. */
do
{
    /* if right child exists, get row from it */
    if (qn->right != NULL)
        rbuff = get_next(qn->right);

    /* else, read the row from the table (the storage handler */
    else
    {
        /*
            Create space for the record buffer and
            store pointer in rbuff
        */
        rbuff = (READ_RECORD *) my_malloc(sizeof(READ_RECORD),
            MYF(MY_ZEROFILL | MY_WME));
        rbuff->rec_buf =
            (byte *) my_malloc(qn->relations[0]->table->s->rec_buff_length,

```

```

MYF(MY_ZEROFILL | MY_WME));

/* check for end of file. Store result in eof array */
qn->eof[1] =
    qn->relations[1]->table->file->rnd_next(rbuff->rec_buf);
if (qn->eof[1] != HA_ERR_END_OF_FILE)
    qn->eof[1] = false;
else
{
    rbuff = NULL;
    qn->eof[1] = true;
}
}
/* if the right buffer is not null, get a new row from table */
if (rbuff != NULL)
{
    /* we need the table information for processing fields */
    if (qn->right == NULL)
        rtable = qn->relations[1]->table;
    else
        rtable = get_table(qn->right);
    if (rtable != NULL)
        memcpy((byte *)rtable->record[0], (byte *)rbuff->rec_buf,
            rtable->s->rec_buff_length);

    /* get the join expression */
    expr = qn->join_expr->get_expression(0);
    Field *cur_field = (Field *)expr->right_op;
    for (Field **field = rtable->field; *field; field++)
        if (strcasecmp((*field)->field_name, cur_field->field_name)==0)
            fright = (*field);

    /*
       If field was found, add the row to the in-memory buffer
       ordered by the join column.
    */
    if ((fright != NULL) && (!fright->is_null()))
        insertion_sort(false, fright, rbuff);
}
} while (rbuff != NULL);
left_record_buffer_ptr = left_record_buff;
right_record_buffer_ptr = right_record_buff;
qn->preempt_pipeline = false;
}
/*
   This is where the actual join code begins.
   We get a tuple from each table and start the compare.
*/

/*

```

```

    if lbuff is null and the left record buffer has data
    get the row from the buffer
*/
if ((lbuff == NULL) && (left_record_buffer_ptr != NULL))
{
    lbuff = left_record_buffer_ptr->record;
    lprev = left_record_buffer_ptr;
    left_record_buffer_ptr = left_record_buffer_ptr->next;
}

/*
    if rbuff is null and the right record buffer has data
    get the row from the buffer
*/
if ((rbuff == NULL) && (right_record_buffer_ptr != NULL))
{
    rbuff = right_record_buffer_ptr->record;
    rprev = right_record_buffer_ptr;
    right_record_buffer_ptr = right_record_buffer_ptr->next;
}

/*
    if the left buffer was null, check to see if a row is
    available from left child.
*/
if (ltable == NULL)
    if (qn->left == NULL)
        ltable = qn->relations[0]->table;
    else
        ltable = get_table(qn->left);

/*
    if the right buffer was null, check to see if a row is
    available from right child.
*/
if (rtable == NULL)
    if (qn->right == NULL)
        rtable = qn->relations[1]->table;
    else
        rtable = get_table(qn->right);

/*
    If there are two rows to compare, copy the record buffers
    to the table record buffers. This transfers the data
    from the internal buffer to the record buffer. It enables
    us to reuse the MySQL code for manipulating fields.
*/
if ((lbuff != NULL) && (rbuff != NULL))
{

```

```

memcpy((byte *)ltable->record[0], (byte *)lbuff->rec_buf,
       ltable->s->rec_buff_length);
memcpy((byte *)rtable->record[0], (byte *)rbuff->rec_buf,
       rtable->s->rec_buff_length);

/* evaluate the join condition */
i = qn->join_expr->compare_join(qn->join_expr->get_expression(0),
                                ltable, rtable);

/* if there is a match...*/
if (i == 0)
{
    /* return the row in the next_tup pointer */
    next_tup = lbuff;

    /* store next rows from buffer (already advanced 1 row) */
    record_buff *left = left_record_buffer_ptr;
    record_buff *right = right_record_buffer_ptr;

    /*
     * Check to see if either buffer needs to be rewound to
     * allow us to process many rows on one side to one row
     * on the other
     */
    check_rewind(left_record_buffer_ptr, lprev,
                 right_record_buffer_ptr, rprev);

    /* set pointer to null to force read on next loop */
    lbuff = NULL;
    rbuff = NULL;

    /*
     * If the left buffer has been changed and if the
     * buffer is not at the end, set the buffer to the next row.
     */
    if (left != left_record_buffer_ptr)
    {
        if (left_record_buffer_ptr != NULL)
        {
            lbuff = left_record_buffer_ptr->record;
        }
    }

    /*
     * If the right buffer has been changed and if the
     * buffer is not at the end, set the buffer to the next row.
     */
    if (right != right_record_buffer_ptr)
    {
        if (right_record_buffer_ptr != NULL)

```



```

    {
        rbuff = right_record_buffer_ptr->record;
    }
}

/* Now check for end of file and save results in eof array */
if (left_record_buffer_ptr == NULL)
    qn->eof[2] = true;
else
    qn->eof[2] = false;
if (right_record_buffer_ptr == NULL)
    qn->eof[3] = true;
else
    qn->eof[3] = false;
}

/* if the rows didn't match...*/
else
{
    /* get next rows from buffers (already advanced) */
    record_buff *left = left_record_buffer_ptr;
    record_buff *right = right_record_buffer_ptr;
    /*
        Check to see if either buffer needs to be rewound to
        allow us to process many rows on one side to one row
        on the other. The results of this rewind must be
        saved because there was no match and we may have to
        reuse one or more of the rows.
    */
    check_rewind(left_record_buffer_ptr, lprev,
        right_record_buffer_ptr, rprev);

    /*
        If the left buffer has been changed and if the
        buffer is not at the end, set the buffer to the next row
        and copy the data into the record buffer/
    */
    if (left != left_record_buffer_ptr)
    {
        if (left_record_buffer_ptr != NULL)
        {
            memcpy((byte *)ltable->record[0],
                (byte *)left_record_buffer_ptr->record->rec_buf,
                ltable->s->rec_buff_length);
            lbuff = left_record_buffer_ptr->record;
        }
    }
}

/*

```

```

        If the right buffer has been changed and if the
        buffer is not at the end, set the buffer to the next row
        and copy the data into the record buffer/
    */
    if (right_record_buffer_ptr != NULL)
        if ((right_record_buffer_ptr->next == NULL) &&
            (right_record_buffer_ptr->prev == NULL))
            lbuff = NULL;
    if (right != right_record_buffer_ptr)
    {
        if (right_record_buffer_ptr != NULL)
        {
            memcpy((byte *)rtable->record[0],
                (byte *)right_record_buffer_ptr->record->rec_buf,
                rtable->s->rec_buff_length);
            rbuff = right_record_buffer_ptr->record;
        }
    }
    /* Now check for end of file and save results in eof array */
    if (left_record_buffer_ptr == NULL)
        qn->eof[2] = true;
    else
        qn->eof[2] = false;
    if (right_record_buffer_ptr == NULL)
        qn->eof[3] = true;
    else
        qn->eof[3] = false;
}
}
else
    next_tup = NULL; /* at end, return null */
break;
}

/* placeholder for exercise... */
case (jnCROSSPRODUCT) :
{
    break;
}
/*
    placeholder for exercises...
    Union and intersect are mirrors of each other -- same code will
    work for both except the dupe elimination/inclusion part (see below)
*/
case (jnUNION) :
case (jnINTERSECT) :
{
    break;
}

```

```

    }
    DEBUG_RETURN(next_tup);
}

```

请注意，除非找到了一个匹配，在任何其他情况下从这些代码返回的记录都将被设置为NULL。这使得get_next()方法里的循环可以反复调用do_join()方法，直到它返回一个匹配为止。这与do_restrict()方法的调用情况很相似。

你也许已经注意到了，我只实现了内联结操作，没有为其他类型的联结操作实现任何代码。这主要是因为我想把那些代码留给读者作为练习（详见本章末尾的练习题）。你将会发现，只要对代码清单12-23里的代码做几处简单的修改就可以让它处理各种外联结操作。至于叉积、并和交操作，可以根据本章前面内容里给出的理论算法来实现它们。

在研究完do_join()方法的伪代码之后，你应该会发现它的源代码比较容易理解了。这个方法里最复杂的部分是check_rewind()方法，这个函数是我为了简化这段代码和提高其可读性而在这个类里实现的。还有其他几个辅助方法，将在接下来的几个小节里对它们做一个简要的说明。

4. 其他方法

在实现DBXP执行引擎的时候，我使用了几个辅助方法。表12-1列出了这些新方法和它们的用途。在接下来的几个小节里，我将对它们当中比较复杂的几个方法进行说明。

表12-1 DBXP查询执行引擎的辅助方法

方 法	说 明
Query_tree::get_next()	从子结点检索出下一条元组
Query_tree::insertion_sort()	创建一个READ_RECORD的排序缓冲区。用途是在联结操作里对输入的元组进行排序
Query_tree::Eof()	为存储引擎或临时缓冲区检查文件尾条件
Query_tree::check_rewind()	检查十分需要对记录缓冲区进行调整，以便再次读入元组寻找多个匹配
send_data()	把数据发送到客户端，见sql_dbxp_parse.cc文件
Expression::evaluate()	为一个限制操作对WHERE子句进行求值
Expression::compare_join()	为一个联结操作对联结条件进行求值
Handler::rnd_init()	对存储引擎进行读操作初始化（详见第7章）
Handler::rnd_next()	从存储引擎读入下一条元组（详见第7章）

● get_next()方法

get_next()方法是DBXP查询执行引擎的核心。它负责调用各种do_...()方法来完成各种查询操作。在执行查询时，它将从DBXP_select_command()方法中的while循环里被调用一次。在完成有关的初始化工作之后，这个方法将执行当前结点上的操作，调用子结点以获得它们的结果。这个过程将以递归方式不断重复，直到当前结点的所有子结点都返回了一条元组为止。代码清单12-24给出了get_next()方法的源代码。

代码清单12-24 get_next()方法

```

/*
    Get the next tuple (row) in the result set.

```

SYNOPSIS

Eof()

query_node *qn IN the operational node in the query tree.

DESCRIPTION

This method is used to get the next READ_RECORD from the pipeline.

The idea is to call prepare() after you've validated the query then call get_next to get the first tuple in the pipeline.

RETURN VALUE

Success = next tuple in the result set

Failed = NULL

*/

READ_RECORD *Query_tree::get_next(query_node *qn)

{

 READ_RECORD *next_tup = NULL;

 int i = 0;

 DEBUG_ENTER("get_next");

/*

For each of the possible node types, perform the query operation by calling the method for the operation. These implement a very high-level abstraction of the operation. The real work is left to the methods.

*/

switch (qn->node_type)

{

 /* placeholder for exercises... */

 case Query_tree::qntDistinct :

 break;

 /* placeholder for exercises... */

 case Query_tree::qntUndefined :

 break;

 /* placeholder for exercises... */

 case Query_tree::qntSort :

 if (qn->preempt_pipeline)

 qn->preempt_pipeline = false;

 break;

/*

For restrict, get a row (tuple) from the table and call the do_restrict method looping until a row is returned (data matches conditions), then return result to main loop in DBXP_select_command.

*/

case Query_tree::qntRestrict :

do

{


```

/* if there is a child, get row from child */
if (qn->left != NULL)
    next_tup = get_next(qn->left);

/* else get the row from the table stored in this node */
else
{
    /* create space for the record buffer */
    if (next_tup == NULL)
        next_tup = (READ_RECORD *) my_malloc(sizeof(READ_RECORD),
                                                MYF(MY_ZEROFILL | MY_WME));

    next_tup->rec_buf =
        (byte *) my_malloc(qn->relations[0]->table->s->rec_buff_length,
                            MYF(MY_ZEROFILL | MY_WME));

    /* read row from table (storage handler */
    qn->eof[0] = qn->relations[0]->table->file->rnd_next(next_tup->rec_buf);

    /* check for end of file */
    if (qn->eof[0] != HA_ERR_END_OF_FILE)
        qn->eof[0] = false;
    else
    {
        qn->eof[0] = true;
        next_tup = NULL;
    }
}

/* if there is a row, call the do_restrict method */
if (next_tup)
    if(!do_restrict(qn, next_tup))
    {
        /* if no row to return, free memory used */
        my_free((gptr)next_tup->rec_buf, MYF(0));
        my_free((gptr)next_tup, MYF(0));
        next_tup = NULL;
    }
} while ((next_tup == NULL) && !Eof(qn));
break;

/*
For project, get a row (tuple) from the table and
call the do_project method. If successful,
return result to main loop in DBXP_select_command.
*/
case Query_tree::qntProject :

    /* if there is a child, get row from child */
    if (qn->left != NULL)
    {

```

```

next_tup = get_next(qn->left);
if (next_tup)
    if (!do_project(qn, next_tup))
    {
        /* if no row to return, free memory used */
        my_free((gptr)next_tup->rec_buf, MYF(0));
        my_free((gptr)next_tup, MYF(0));
        next_tup = NULL;
    }
}

/* else get the row from the table stored in this node */
else
{
    /* create space for the record buffer */
    if (next_tup == NULL)
        next_tup = (READ_RECORD *) my_malloc(sizeof(READ_RECORD),
                                                MYF(MY_ZEROFILL | MY_WME));

    next_tup->rec_buf =
        (byte *)my_malloc(qn->relations[0]->table->s->rec_buff_length + 20,
                           MYF(MY_ZEROFILL | MY_WME));

    /* read row from table (storage handler */
    qn->eof[0] = qn->relations[0]->table->file->rnd_next(next_tup->rec_buf);

    /* check for end of file */
    if (qn->eof[0] != HA_ERR_END_OF_FILE)
        qn->eof[0] = false;
    else
    {
        qn->eof[0] = true;
        next_tup = NULL;
    }

    /* if there is a row, call the do_project method */
    if (next_tup)
        if (!do_project(qn, next_tup))
        {
            /* no row to return, free memory used */
            my_free((gptr)next_tup->rec_buf, MYF(0));
            my_free((gptr)next_tup, MYF(0));
            next_tup = NULL;
        }
    }
    break;
}

/*
For join, loop until either a row is returned from the
do_join method or we are at end of file for both tables.

```

```

    If successful (data matches conditions),
    return result to main loop in DBXP_select_command.
*/
case Query_tree::qntJoin :
    do
    {
        if (next_tup)
        {
            /* if no row to return, free memory used */
            my_free((gptr)next_tup->rec_buf, MYF(0));
            my_free((gptr)next_tup, MYF(0));
            next_tup = NULL;
        }
        next_tup = do_join(qn);
    }
    while ((next_tup == NULL) && !Eof(qn));
    break;
}
DEBUG_RETURN(next_tup);
}

```

● send_data()方法

send_data()方法是一个辅助方法，它将调用MySQL代码里负责处理通信问题的Protocol类来把数据输出到客户端。这个方法是从MySQL源代码里借来的，我稍微修改了几个地方以实现DBXP执行引擎那（相对）简单的执行过程。这个方法将调用Item超类的item_send()方法把字段值发送给客户端。代码清单12-25给出了send_data()方法的源代码。

代码清单12-25 send_data()方法

```

/*
Send data

SYNOPSIS
    send_data()
    Protocol *p      IN the Protocol class
    THD *thd         IN the current thread
    List<Item> *items IN the list of fields identified in the row
DESCRIPTION
    This method sends the data to the client using the protocol class.

RETURN VALUE
    Success = 0
    Failed = 1
*/
bool send_data(Protocol *protocol, List<Item> &items, THD *thd)
{
    DEBUG_ENTER("send_data");

    /* use a list iterator to loop through items */

```

```

List_iterator_fast<Item> li(items);

char buff[MAX_FIELD_WIDTH];
String buffer(buff, sizeof(buff), &my_charset_bin);

/* this call resets the transmission buffers */
protocol->prepare_for_resend();

/* for each item in the list (a field), send data to the client */
Item *item;
while ((item=li++))
{
    /*
     * Use the MySQL send method for the item class to write to network.
     * If unsuccessful, free memory and send error message to client.
     */
    if (item->send(protocol, &buffer))
    {
        protocol->free();          /* Free used buffer
        my_message(ER_OUT_OF_RESOURCES, ER(ER_OUT_OF_RESOURCES), MYF(0));
        break;
    }
}
/* increment row count */
thd->sent_row_count++;

/* if network write was ok, return */
if (!thd->vio_ok())
    DEBUG_RETURN(0);
/* write failed, return error code to client */
if (!thd->net.report_error)
    DEBUG_RETURN(protocol->write());
/* remove last row from buffer for error processing */
protocol->remove_last_row();
DEBUG_RETURN(1);
}

```

这个方法使用了item类，它将调用send()方法并把一个指针传递到protocol类的一个实例。一个字段项的值就是这样被发送到客户端去的。send_data()方法也是投影和联结操作所涉及的列接受处理的地方。这个部分是MySQL源代码里最让人感到好奇的地方之一。MySQL类是怎样知道应该把哪些列发送给客户端的？这还得回到build_query_tree()方法说起。还记得select_lex类里有一个被识别出来的字段清单吗？DBXP代码在下面这条语句里捕获了这些字段。这个清单是直接来自SELECT命令的列清单并由解析器代码填充的。

```
qt->result_fields = lex->select_lex.item_list;
```

这些字段是在线程的扩展结构里捕获的。MySQL代码将把这个字段清单里出现的任何数据写出去。

● check_rewind()方法

这个方法是联结算法的一部分，但很多数据库书籍对它只是一笔带过，没有详细讨论。这个方法

负责调整用来存放来自表的行的缓冲区，好让算法能够再次使用已经被处理过的行。这很有必要，因为第一个表里一个行有可能匹配第二个表里的多个行。这个方法所实现的概念并不复杂，但通过编程来实现这个概念还真不容易。还好，我替你们解决了这个麻烦。

这个方法是通过检查记录缓冲区里的行来工作的。它的输入参数是两个分别指向当前左、右记录缓冲区的指针和两个分别指向上次调用时的左、右记录缓冲区的指针。记录缓冲区被实现为一个双向链表，这使我们可以在缓冲区里快速地向后或向前移动到下一条记录。

为了让数据源源不断地流向do_join()方法，这个方法必须处理几种条件。这些条件是在找到一个匹配记录之后，对联结条件进行求值的结果。“没有找到匹配”的情况是在do_join()方法里处理的，不归check_rewind()方法管。

- ❑ 如果左缓冲区里的下一条记录是右缓冲区的一个匹配，回卷右缓冲区直到右缓冲区里的联结条件少于左缓冲区为止。
- ❑ 如果左缓冲区里的下一条记录不是右缓冲区的一个匹配，把右缓冲区设置为上一次的右记录指针。
- ❑ 如果左缓冲区已经到达末尾但右缓冲区里还有记录，并且如果上一次的左记录指针的值是右记录指针的一个匹配，把左记录指针设置为上一次的左记录指针。

这个方法显然更重视左记录缓冲区。换句话说，这个方法以左缓冲区为主，让右缓冲区与左缓冲区保持同步（也就是我们前面提到过的“左优先”策略）。代码清单12-26给出了check_rewind()方法的源代码。

代码清单12-26 check_rewind()方法

```
/*
  Adjusts pointers to record buffers for join.

  SYNOPSIS
    check_rewind()
    record_buff *cur_left IN the left record buffer
    record_buff *cur_left_prev IN the left record buffer previous
    record_buff *cur_right IN the left record buffer
    record_buff *cur_right_prev IN the left record buffer previous

  DESCRIPTION
    This method is used to push a tuple back into the buffer
    during a join operation that preempts the pipeline.

  NOTES
    Now, here's where we have to check the next tuple in each
    relation to see if they are the same. If one of them is the
    same and the other isn't, push one of them back.

    We need to rewind if one of the following is true:
    1. The next record in R2 has the same join value as R1
    2. The next record in R1 has the same join value as R2
    3. The next record in R1 has the same join value and R2 is
       different (or EOF)
```

4. The next record in R2 has the same join value and R1 is different (or EOF)

```

RETURN VALUE
    Success = int index number
    Failed = -1
*/
int Query_tree::check_rewind(record_buff *cur_left,
                             record_buff *curr_left_prev,
                             record_buff *cur_right,
                             record_buff *curr_right_prev)
{
    record_buff *left_rcd_ptr = cur_left;
    record_buff *right_rcd_ptr = cur_right;
    int i;
    DEBUG_ENTER("check_rewind");

    /*
        If the next tuple in right record is the same as the present tuple
        AND the next tuple in right record is different, rewind until
        it is the same
    else
        Push left record back.
    */

    /* if both buffers are at EOF, return -- nothing to do */
    if ((left_rcd_ptr == NULL) && (right_rcd_ptr == NULL))
        DEBUG_RETURN(0);

    /* if the currently processed record is null, get the one before it */
    if (cur_right == NULL)
        right_rcd_ptr = curr_right_prev;

    /*
        if left buffer is not at end, check to see
        if we need to rewind right buffer
    */
    if (left_rcd_ptr != NULL)
    {
        /* compare the join conditions to check order */
        i = memcmp(left_rcd_ptr->field_ptr, right_rcd_ptr->field_ptr,
                   left_rcd_ptr->field_length < right_rcd_ptr->field_length ?
                   left_rcd_ptr->field_length : right_rcd_ptr->field_length);

        /*
            i == 0 means the rows are the same. In this case, we need to
            check to see if we need to advance or rewind the right buffer.
        */
        if (i == 0)
        {

```

```

/*
    If there is a next row in the right buffer, check to see
    if it matches the left row. If the right row is greater
    than the left row, rewind the right buffer to one previous
    to the current row or until we hit the start.
*/
if (right_rcd_ptr->next != NULL)
{
    right_rcd_ptr = right_rcd_ptr->next;
    i = memcmp(left_rcd_ptr->field_ptr, right_rcd_ptr->field_ptr,
        left_rcd_ptr->field_length < right_rcd_ptr->field_length ?
        left_rcd_ptr->field_length : right_rcd_ptr->field_length);
    if (i > 0)
    {
        do
        {
            if (right_rcd_ptr->prev != NULL)
            {
                right_rcd_ptr = right_rcd_ptr->prev;
                i = memcmp(left_rcd_ptr->field_ptr, right_rcd_ptr->field_ptr,
                    left_rcd_ptr->field_length < right_rcd_ptr->field_length ?
                    left_rcd_ptr->field_length : right_rcd_ptr->field_length);
            }
        }
        while ((i == 0) && (right_rcd_ptr->prev != NULL));

        /* now advance one more to set pointer to correct location */
        if (right_rcd_ptr->next != NULL)
            right_rcd_ptr = right_rcd_ptr->next;
    }
    /* if no next right row, rewind to previous row */
    else
        right_rcd_ptr = right_rcd_ptr->prev;
}
/*
    If there is a next row in the left buffer, check to see
    if it matches the right row. If there is a match and the right
    buffer is not at start, rewind the right buffer to one previous
    to the current row.
*/
else if (left_rcd_ptr->next != NULL)
{
    if (right_rcd_ptr->prev != NULL)
    {
        i = memcmp(left_rcd_ptr->field_ptr, right_rcd_ptr->prev->field_ptr,
            left_rcd_ptr->field_length < right_rcd_ptr->prev->field_length ?
            left_rcd_ptr->field_length : right_rcd_ptr->prev->field_length);
    }
    if ((i == 0) && (right_rcd_ptr->prev != NULL))
        right_rcd_ptr = right_rcd_ptr->prev;
}

```

```

    }
}
/* if the left row is less than right row, rewind right buffer */
else if (i < 0)
{
    if (right_rcd_ptr->prev != NULL)
        right_rcd_ptr = right_rcd_ptr->prev;
}
/* if the right row is less than the left row, advance right row */
else
{
    if (right_rcd_ptr->next != NULL)
        right_rcd_ptr = right_rcd_ptr->next;
}
}
/*
Rows don't match so advance the right buffer and check match again.
if they still match, rewind left buffer.
*/
else
{
    if (right_rcd_ptr->next != NULL)
    {
        i = memcmp(curr_left_prev->field_ptr, right_rcd_ptr->field_ptr,
            curr_left_prev->field_length < right_rcd_ptr->field_length ?
            curr_left_prev->field_length : right_rcd_ptr->field_length);
        if (i == 0)
            left_rcd_ptr = curr_left_prev;
    }
}
/* set buffer pointers to adjusted rows from buffers */
left_record_buffer_ptr = left_rcd_ptr;
right_record_buffer_ptr = right_rcd_ptr;
DEBUG_RETURN(0);
}

```

对DBXP查询执行引擎源代码的分析到这里也就差不多了。接下来是编译和测试它的时间了。

12.2.4 代码的编译和测试

如果你还没有下载过本章的源代码，请下载它们并把它们放到MySQL源代码树根目录下的/sql子目录里。你应该花点儿时间浏览一下那些代码，让自己对那些方法做到心中有数。这样，万一你需要调试这些代码去适应你的系统配置，或者如果你想再添加其他加强的功能或做练习题，就不至于临时抱佛脚了。把所有的源代码文件都下载回来并浏览过它们之后，你还需要把这些文件添加到你的制作文件（Linux平台）或项目文件（Windows平台）里。

提示 把源代码文件添加到DBXP项目里并编译它们的详细步骤见第11章。

在安装并编译好新代码后，你就可以启动你的MySQL服务器并运行测试了。可以运行我们在本章开头创建的测试，也可以找个MySQL命令行客户端工具以手动方式输入那些SQL命令。代码清单12-27给出了一个测试过程示例。

代码清单12-27 测试DBXP查询执行引擎

```
mysql> SELECT DBXP first_name, last_name, sex, id FROM staff;
```

first_name	last_name	sex	id
Bill	Smith	M	333445555
William	Walters	M	123763153
Alicia	St.Cruz	F	333444444
Goy	Hong	F	921312388
Rajesh	Kardakarna	M	800122337
Monty	Smythe	M	820123637
Richard	Jones	M	830132335
Edward	Engles	M	333445665
Beware	Borg	F	123654321
Wilma	Maxima	F	123456789

10 rows in set (0.01 sec)

```
mysql> SELECT DBXP id FROM staff;
```

id
333445555
123763153
333444444
921312388
800122337
820123637
830132335
333445665
123654321
123456789

10 rows in set (0.00 sec)

```
mysql> SELECT DBXP dir_name FROM directorate;
```

dir_name
Development
Human Resources

```
| Management |
+-----+
3 rows in set (0.00 sec)
```

```
mysql> SELECT DBXP id, dir_name FROM staff
mysql> JOIN directorate ON staff.mgr_id = directorate.dir_head_id;
```

```
+-----+-----+
| id      | dir_name |
+-----+-----+
| 123763153 | Human Resources |
| 921312388 | Human Resources |
| 333445555 | Management      |
| 123654321 | Management      |
| 800122337 | Development      |
| 820123637 | Development      |
| 830132335 | Development      |
| 333445665 | Development      |
| 123456789 | Development      |
+-----+-----+
9 rows in set (0.01 sec)
```

```
mysql> SELECT DBXP * FROM staff WHERE staff.id = '123456789';
```

```
+-----+-----+-----+-----+-----+-----+-----+
| id      | first_name | mid_name | last_name | sex | salary | mgr_id |
+-----+-----+-----+-----+-----+-----+-----+
| 123456789 | Wilma      | N        | Maxima    | F   | 43000  | 333445555 |
+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT DBXP first_name, last_name FROM staff JOIN directorate
WHERE staff.mgr_id = directorate.dir_head_id and directorate.dir_code = 'N41';
```

```
+-----+-----+
| first_name | last_name |
+-----+-----+
| Rajesh     | Kardakarna |
| Monty      | Smythe     |
| Richard    | Jones      |
| Edward     | Engles     |
| Wilma      | Maxima     |
+-----+-----+
5 rows in set (0.01 sec)
```

```
mysql> SELECT DBXP * FROM directorate
mysql> JOIN building ON directorate.dir_code = building.dir_code;
```

```
+-----+-----+-----+-----+-----+
| dir_code | dir_name | dir_head_id | dir_code | building |
+-----+-----+-----+-----+-----+
```

M00	Management	333444444	M00	1000
N01	Human Resources	123654321	N01	1453
N41	Development	333445555	N41	1300
N41	Development	333445555	N41	1301
N41	Development	333445555	N41	1305

5 rows in set (0.01 sec)

```
mysql> SELECT DBXP directorate.dir_code, dir_name, building, dir_head_id
mysql> FROM directorate JOIN building
mysql> ON directorate.dir_code = building.dir_code;
```

dir_code	dir_name	building	dir_head_id
M00	Management	1000	333444444
N01	Human Resources	1453	123654321
N41	Development	1300	333445555
N41	Development	1301	333445555
N41	Development	1305	333445555

5 rows in set (0.01 sec)

mysql>

12.3 小结

本章解释了数据库内部的查询执行操作，讲解了如何扩展查询树的概念以创建一种利用树结构来执行查询的查询执行引擎。掌握了这些技术和知识，你就可以进一步了解DBXP引擎以及如何使用它去学习更高深的数据库技术。

现在已经到了本书的末尾，也许你正在想下一步该做些什么。本书的第三部分为你提供了一个基于MySQL的实验性查询引擎，你可以通过它来探讨自己的内部数据库技术的实现。最重要的是，你可以按照自己的想法去改造DBXP代码。也许你只是想做些实验，但也许你还想实现并和交操作，甚至也许你还想扩展DBXP引擎以实现MySQL的全部查询功能。不管你想用在本书第三部分里所学到的知识干什么，总是可以通过为MySQL实现一个新的查询引擎而让朋友和同事们刮目相看！

练习

下面是几个值得进一步研究的问题。它们在同类问题中很有代表性，在研究关系数据库技术时，可以把它们作为实验题材（或课堂作业）。

1. 请完成do_join()方法的代码，使它支持MySQL所支持的所有联结类型。提示：必须在开始优化前确定联结操作的类型。请到MySQL解析器的源代码里找答案。
2. 请仔细阅读Query_tree类中的check_rewind()方法的源代码。把这个实现改成使用临时表，这可以避免对大表进行联结操作时消耗大量的内存。

3. 对DBXP查询引擎的性能进行评估。多进行几次测试并记下执行时间。把这些结果与使用MySQL查询引擎执行同样查询时的测试结果进行比较。DBXP引擎与MySQL相比如何？
4. 为交操作剔除重复的操作为什么是不必要的？有没有必须进行这种剔除操作的情况？如果有，它们是什么？
5. (高级) MySQL目前还不支持叉积或交操作（按照C. J. Date给出的定义）。请修改MySQL解析器使它能够识别这些新关键字并处理像SELECT * FROM A CROSS B和SELECT * FROM A INTERSECT B这样的查询命令，并且为DBXP执行引擎增加这些功能。提示：阅读do_join()方法的源代码。
6. (高级) 请创建一个更完备的测试查询列表以了解DBXP引擎都有哪些不足。如果你想扩展DBXP引擎的能力，需要做哪些修改？



附录

本附录由3个部分组成：本书用到的参考书目、本书例子里使用的样板数据库的描述以及如何解决第10~12章的练习题的提示。

参考书目

以下书目有些是我在编写本书时参考的文献资料，有些可以帮助你进一步学习和研究数据库技术。这份清单是按题材组织的。

数据库理论

- Belussi, A. , E. Bertino, and B. Catania. 1998. An Extended Algebra for Constraint Databases. *IEEE Transactions on Knowledge and Data Engineering* 10(5): 686–705.
- Date, C. J. and H. Darwen. 2000. *Foundation for Future Database Systems: The Third Manifesto*. Reading, MA: Addison-Wesley.
- Date, C. J. 2001. *The Database Relational Model: A Retrospective Review and Analysis*. Reading, MA: Addison-Wesley.
- Elmasri, R. and S. B. Navathe. 2003. *Fundamentals of Database Systems*, 4th ed. Boston: Addison-Wesley.
- Franklin, M. J. , B. T. Jonsson, and D. Kossmann. 1996. Performance Tradeoffs for Client-Server Query Processing. *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, 149–160.
- Gassner, P., G. M. Lohman, K. B. Schiefer, and Y. Wang. 1993. Query Optimization in the IBM DB2 Family. *Bulletin of the Technical Committee on Data Engineering* 16(4): 4–17.
- Ioannidis, Y. E., R. T. Ng, K. Shim, and T. Sellis. 1997. Parametric Query Optimization. *VLDB Journal* 6:132–151.
- Kossman, D. and K. Stocker. 2000. Iterative Dynamic Programming: A New Class of Query Optimization Algorithms. *ACM Transactions on Database Systems* 25(1): 43–82.
- Lee, C., C. Shih, and Y. Chen. 2001. A Graph-Theoretic Model for Optimizing Queries Involving Methods. *VLDB Journal* 9: 327–343.
- Selinger, P. G., M. M. Astraham, D. D. Chamberlin, R. A. Lories, and T. G. Price. 1979. Access Path Selection in a Relational Database Management System. *Proceedings of the ACM SIGMOD*

International Conference on the Management of Data, Aberdeen, Scotland, 23–34.

Stonebraker, M., E. Wong, P. Kreps. 1976. The Design and Implementation of INGRES. *ACM Transactions on Database Systems* 1(3): 189–222.

Stonebraker, M. and J. L. Hellerstein. 1998. *Readings in Database Systems*, 3rd ed. San Mateo, CA: Morgan Kaufmann Publishers.

Tucker, A. B. 2004. *Computer Science Handbook*, 2nd ed. Boca Raton, FL: CRC Press.

Werne, B. 2001. *Inside the SQL Query Optimizer*. Progress Worldwide Exchange 2001, Washington, DC: www.peg.com/techpapers/2001Conf/.

通用

Rosenberg, D., M. Stephens, and M. Collins-Cope. 2005. *Agile Development with ICONIX Process*. Berkeley, CA: Apress.

MySQL

Burgelman, R.A., A. S. Grove, and P. E. Meza. 2006. *Strategic Dynamics*. New York: McGraw-Hill.

Kruckenberg, M. and J. Pipes. 2005. *Pro MySQL*. Berkeley, CA: Apress.

开源

Paulson, J. W. 2004. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions on Software Engineering*, 30(5): 246–256.

网站

www.opensource.org—Open Source Initiative (OSI)

<http://dev.mysql.com>—MySQL's Developer Zone

www.mysql.com/company/legal/licensing/opensource-license.html—MySQL Open Source License

www.gnu.org/licenses/gpl.html—The GNU General Public License

www.mysql.com/support/community_support.html—MySQL support options

www.bitkeeper.com—BitKeeper

www.activestate.com—ActivePerl for Windows

<http://jeremy.zawodny.com/mysql/mytop>—mytop for MySQL

www.gnu.org/software/diffutils/diffutils.html—Diffutils for Linux

www.gnu.org/software/patch/—patch (GNU Project)

www.gnu.org/software/gdb/documentation—GDB: The GNU Project Debugger

<ftp://www.gnu.org/software/ddd>—GNU Data Display Debugger

<http://undo-software.com>—Undo Software

<http://forums.mysql.com>—MySQL Forums

<http://lists.mysql.com>—MySQL Lists

<http://gnuwin32.sourceforge.net/packages/bison.htm>—Bison

www.dinosaur.compilertools.net—Lex and YACC

www.postgresql.org/—PostgreSQL

样板数据库

代码清单A-1是本书最后几章中的例子所使用的样板数据库的代码，展示了数据库的SQL转储。

代码清单A-1 样板数据库的创建语句

```
-- MySQL dump 10.10
--
-- Host: localhost    Database: expert_mysql
-- -----
-- Server version      5.1.9-beta-debug-DBXP 1.0

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40103 SET @OLD_TIME_ZONE=@@TIME_ZONE */;
/*!40103 SET TIME_ZONE='+00:00' */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0
*/;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

CREATE DATABASE IF NOT EXISTS expert_mysql;

--
-- Table structure for table `expert_mysql`.`building`
--

DROP TABLE IF EXISTS `expert_mysql`.`building`;
CREATE TABLE `expert_mysql`.`building` (
  `dir_code` char(4) NOT NULL,
  `building` char(6) NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `expert_mysql`.`building`
--

/*!40000 ALTER TABLE `expert_mysql`.`building` DISABLE KEYS */;
LOCK TABLES `expert_mysql`.`building` WRITE;
INSERT INTO `expert_mysql`.`building` VALUES
('N41','1300'),
('N01','1453'),
```

```
('M00','1000'),
('N41','1301'),
('N41','1305');
UNLOCK TABLES;
/*!40000 ALTER TABLE `expert_mysql`.`building` ENABLE KEYS */;

--
-- Table structure for table `expert_mysql`.`directorate`
--

DROP TABLE IF EXISTS `expert_mysql`.`directorate`;
CREATE TABLE `expert_mysql`.`directorate` (
  `dir_code` char(4) NOT NULL,
  `dir_name` char(30) DEFAULT NULL,
  `dir_head_id` char(9) DEFAULT NULL,
  PRIMARY KEY (`dir_code`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `expert_mysql`.`directorate`
--

/*!40000 ALTER TABLE `expert_mysql`.`directorate` DISABLE KEYS */;
LOCK TABLES `expert_mysql`.`directorate` WRITE;
INSERT INTO `expert_mysql`.`directorate` VALUES
('N41','Development','333445555'),
('N01','Human Resources','123654321'),
('M00','Management','333444444');
UNLOCK TABLES;
/*!40000 ALTER TABLE `directorate` ENABLE KEYS */;

--
-- Table structure for table `expert_mysql`.`staff`
--

DROP TABLE IF EXISTS `expert_mysql`.`staff`;
CREATE TABLE `expert_mysql`.`staff` (
  `id` char(9) NOT NULL,
  `first_name` char(20) DEFAULT NULL,
  `mid_name` char(20) DEFAULT NULL,
  `last_name` char(30) DEFAULT NULL,
  `sex` char(1) DEFAULT NULL,
  `salary` int(11) DEFAULT NULL,
  `mgr_id` char(9) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `expert_mysql`.`staff`
--
```



```
/*!40000 ALTER TABLE `expert_mysql`.`staff` DISABLE KEYS */;
LOCK TABLES `expert_mysql`.`staff` WRITE;
INSERT INTO `expert_mysql`.`staff` VALUES
('333445555','John','Q','Smith','M',30000,'333444444'),
('123763153','William','E','Walters','M',25000,'123654321'),
('333444444','Alicia','F','St.Cruz','F',25000,NULL),
('921312388','Goy','X','Hong','F',40000,'123654321'),
('800122337','Rajesh','G','Kardakarna','M',38000,'333445555'),
('820123637','Monty','C','Smythe','M',38000,'333445555'),
('830132335','Richard','E','Jones','M',38000,'333445555'),
('333445665','Edward','E','Engles','M',25000,'333445555'),
('123654321','Beware','D','Borg','F',55000,'333444444'),
('123456789','Wilma','N','Maxima','F',43000,'333445555');
UNLOCK TABLES;
/*!40000 ALTER TABLE `expert_mysql`.`staff` ENABLE KEYS */;

--
-- Table structure for table `tasking`
--

DROP TABLE IF EXISTS `expert_mysql`.`tasking`;
CREATE TABLE `expert_mysql`.`tasking` (
  `id` char(9) NOT NULL,
  `project_number` char(9) NOT NULL,
  `hours_worked` double DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `tasking`
--

/*!40000 ALTER TABLE `tasking` DISABLE KEYS */;
LOCK TABLES `expert_mysql`.`tasking` WRITE;
INSERT INTO `expert_mysql`.`tasking` VALUES
('333445555','405',23),
('123763153','405',33.5),
('921312388','601',44),
('800122337','300',13),
('820123637','300',9.5),
('830132335','401',8.5),
('333445555','300',11),
('921312388','500',13),
('800122337','300',44),
('820123637','401',500.5),
('830132335','400',12),
('333445665','600',300.25),
('123654321','607',444.75),
('123456789','300',1000);
```

```

UNLOCK TABLES;
/*!40000 ALTER TABLE `expert_mysql`.`tasking` ENABLE KEYS */;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;

```

第 10 章至第 12 章练习题注解

本节为第10章至第12章末尾的练习题给出了一些提示和有帮助的指导。有些习题是实践题，它们的解决方案往往很长，难以收录在这篇附录里。对于那些需要编程解决的问题，我给出了一些关于如何为解决方案编写代码的提示。对于其他问题，我提供了一些可以帮助大家完成练习的有用信息。

第 10 章

以下问题来自第10章。

1. 图10-1里的查询命令暴露出了某个表的设计方案有缺陷。你能发现这个缺陷吗？这个缺陷违反了某个范式吗？如果是，那么是哪一种范式？

请注意semester属性。这项数据代表了多少个值？当你需要访问这个属性（或字段）的某个部分时，这样的数据会导致性能低下的查询。比如说，如果想查询2001年所有的学期，将不得不在WHERE子句里使用LIKE操作符：WHERE semester LIKE '%2001'。这种把多个数据以某种编码的形式打包在同一个字段（术语称之为“多值字段”）里的做法违反了第一范式。

2. 剖析TABLE结构并把SELECT DBXP存根改成可以返回关于表及其字段的信息。

请参考第8章里的show_disk_usage_command()方法修改SELECT DBXP命令的代码实现，让它们返回关于表及其字段的信息。这次需要包括关于表的元数据。提示：好好研究一下table类。

3. 修改EXPLAIN SELECT DBXP命令，让它产生与MySQL的EXPLAIN SELECT命令类似的输出信息。

修改EXPLAIN SELECT命令的代码实现，让它们生成像MySQL版EXPLAIN命令那样的信息。注意，你需要在Query_tree类里添加一个新方法来收集关于被优化的查询命令的信息。

4. 修改build_query_tree()函数，让它可以识别并处理LIMIT子句。

你需要添加一些代码来识别SELECT命令里的LIMIT子句并根据该子句对结果集里的记录个数进行限制。提示：下面的代码可以捕获LIMIT子句的值。你将需要修改DBXP_select_command()方法里的代码来处理这个操作的其余动作。

```

SELECT_LEX_UNIT *unit= &lex->unit;
unit->set_limit(unit->global_parameters);

```

5. 如何修改query_node结构才能让它支持HAVING、GROUP BY和ORDER BY子句？

最好的设计方案是坚持查询树概念的设计方案，可以考虑把这些子句分别表示为查询树里的一个结点。也可以考虑是否有一些启发可以应用于这些操作。提示：在最接近叶结点的地方处理HAVING子

句是否更有效率？最后，请考虑是否应该增加一些规则来限制这些结点中的每一种在查询树里的最大个数。

第 11 章

以下问题来自第11章。

1. 请完成`balance_joins()`方法的代码。提示：你需要创建一个算法来移动那些在查询树里上下相邻的联结，让限制性最强的联结操作最先执行（位于查询树的最低层）。

这道题需要你想办法在查询树里移动那些“联结”结点，并把限制性最强的“联结”结点向下推。解决问题的关键是如何利用关于表的统计信息去判断哪些联结操作将产生最少的结果。请在`handler`类和`table`类的源代码里寻找访问这些统计数据的方法。此外，你还需要一些辅助方法遍历查询树来收集关于表的信息。这很有必要，因为联结操作有可能出现在查询树的高层结点里，不直接引用有关的表。

2. 请完成`cost_optimization()`方法的代码。提示：你需要遍历查询树以找出哪些结点可以使用索引。

解答这道题的关键是利用`handler`类和`table`类提供的信息和手段来确定哪些表有索引，被索引的又是哪几列。

3. 请仔细阅读DBXP启发式优化器的代码。它能覆盖所有可能的查询情况吗？如果不能，还需要增加哪些规则才能让它做到这一点。

你应该发现还有许多这样的好规则，这种优化器只是实现了最有效的几条启发。比如说，你还可以为它再实现一些把GROUP BY和HAVING子句也考虑进来的规则。在编写用来实现那些新规则的方法时，可以参考我为投影和限制操作实现的`do_project()`和`do_restrict()`方法。

4. 请仔细阅读查询树和DBXP启发式优化器的代码。怎样才能实现查询树类里的DISTINCT结点类型？提示：阅读`heuristic_optimization()`方法里紧跟在`prune_tree()`方法后面的代码。

解答这道题的大部分线索都在样板代码里。紧跟在`prune_tree()`方法后面的代码可以让你知道如何识别查询命令里的DISTINCT选项。

5. 如何修改代码才能让它识别出非法的查询？判断一个查询是否非法的条件是什么？你又该如何测试它们？

解答这道题需要完成的工作已经由MySQL替你完成了一大半。比如说，MySQL解析器可以把包含语法错误的查询语句识别出来，并返回一条相应的出错消息。不过，为了对付那些语法正确但语义有错误的查询，你将需要增加一些出错处理代码来检查各种异常情况。比如说，试试一个语法正确但引用的是在表里不存在的列的查询。请为这类错误情况创建一些测试并跟踪（或调试）有关的代码，这可以帮助你发现应该把新的出错处理代码添加到什么地方。最后，你还可以在`Query_tree`类里创建一个新方法去检查查询树本身——如果你想创建一些新的结点类型或是实现一些新的启发式方法的话，有一个这样的方法将非常方便。

6. （高级）MySQL目前还不支持交操作（按照C. J. Date给出的定义）。请修改MySQL解析器，让它能够识别新关键字INTERSECT并处理像SELECT * FROM A INTERSECT B这样的查询。这种操作有什么局限性？处理它们是否需要修改优化器？

这道题听起来很难，但它的解决方案相当直截了当。请考虑创建这样一种新的结点类型：它的名字是intersect，有两个子结点，结点上的操作是返回那些同时出在两个表里的行。提示：你可以选用一种归并排序算法（这样的算法有很多）来完成这项任务。

7.（高级）如果让你来实现HAVING、GROUP BY和ORDER BY子句，你会怎么做？请修改DBXP优化器使它支持这些子句。

有许多办法可以完成这个任务。为了与Query_tree类的设计思路保持一致，你应该把这些操作中的每一种分别表示为一种新的结点类型，然后为每一种新的结点类型编写一个方法来实现相应的操作，就像对限制、投影和联结操作进行处理那样。需要提醒的是，HAVING子句几乎总是与GROUP BY子句或ORDER BY子句同时出现，而你应该把它们分别表示为一个结点。HAVING子句通常放在最后处理。

第 12 章

以下问题来自第12章。

1. 请完成do_join()方法的代码，使它支持MySQL所支持的所有联结类型。提示：必须在开始优化前确定联结操作的类型。请到MySQL解析器的源代码里找答案。

为了完成这道题，你可能需要调整do_join()方法的整体结构。我在实现do_join()方法时是把所有的代码都放在了一起，但更精巧的解决方案应该是在do_join()方法里使用一个switch-case语句并通过它的各个case分支为每一种不同类型的联结操作调用一个辅助方法。你还可以编写一些其他的辅助方法来完成各种通用的操作（参见preempt_pipeline代码）。用来实现其他类型的联结操作的代码与第12章里的do_join()方法将非常相似。

2. 请仔细阅读Query_tree类中的check_rewind()方法的源代码。把这个实现改成使用临时表，这可以避免对大表进行联结操作时消耗大量的内存。

这道题的解决方案也很简单明了。创建一个临时表的细节可以在sql_select.cc文件中的MySQL代码里找到。提示：请仔细研读sql_select.cc文件中的create_table()和insert()方法，需要你编写的代码和它们大同小异。你还可以使用Spartan基类并创建一个临时表来存放记录缓冲区。

3. 对DBXP查询引擎的性能进行评估。多进行几次测试并记下执行时间。把这些结果与使用MySQL查询引擎执行同样查询时的测试结果进行比较。DBXP引擎与MySQL相比如何？

记录执行时间的办法有很多。你可以找个秒表以人工方式记录执行时间，也可以增加一些代码去捕获系统时间。我建议你采用后一种办法，因为这是在测计算机软件的相对速度，所以需要一种既可行又可靠的办法。我之所以用了“相对”这个词，是因为有许多环境因素（系统配置、测试时还运行着哪些软件等）会影响你的测试结果。在进行这种测试时，一定要多进行几次测试并对测试结果进行统计分析。这样你就会得到一组标准化的数据用于比较。

4. 为交操作剔除重复的操作为什么是不必要的？有没有必须进行这种剔除操作的情况？如果有，它们是什么？

要想回答这个问题，必须先把什么是交操作搞清楚。这里对“交操作”的定义是：返回同时出现在各有关表（交操作并非仅限于两个表）里的所有行。如果各表本身没有彼此重复，对它们进行交操作而得到的结果集里就不可能有重复。可是，如果那些表来自查询树里的底层结点，此前的操作都没有剔除重复，查询命令里也没有包括求异操作，就需要在交操作里剔除那些重复了。总之，这个问题

的答案是“随机应变”。

5. (高级) MySQL目前还不支持叉积或交操作(按照C. J. Date给出的定义)。请修改MySQL解析器使它能够识别这些新关键字并处理像SELECT * FROM A CROSS B和SELECT * FROM A INTERSECT B这样的查询, 并且为DBXP执行引擎增加这些功能。提示: 阅读do_join()方法的源代码。

你需要修改的文件与我们在添加DBXP关键字时修改的文件是一样的, 包括lex.h和sql_yacc.yy文件。你可能需要扩展sql_lex结构, 让它能够识别和记录新的操作类型。

6. (高级) 请创建一个更完备的测试查询列表以了解DBXP引擎都有哪些不足。如果你想扩展DBXP引擎的能力, 需要做哪些修改?

首先, 需要扩展查询树以便包括HAVING、GROUP BY和ORDER BY子句。你还应该考虑为它添加处理各种聚集函数(max()、min()等)的能力。这些聚集函数可以放在Expression类里实现, 你将需要编写一些新方法来解析和评估这些聚集函数。



Expert MySQL

深入理解MySQL

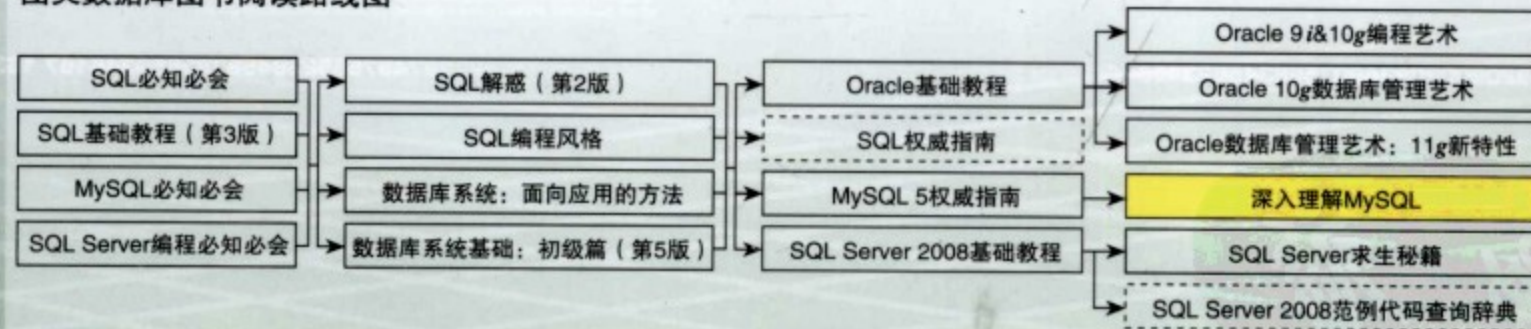
MySQL是目前最流行的开源数据库，经过多年发展，日趋成熟，已经能够和主流的商业数据库相抗衡。

本书结合MySQL源代码深入讲解了MySQL数据库的核心知识。全书分为三个部分，从介绍数据库基础知识开始，逐步深入到存储引擎，最后介绍了查询优化器等数据库内部结构。第三部分还提供了一些有关数据库的实验，以便读者亲自动手来构建一个实验性质的数据库，从而加深对数据库内部结构的了解。作者很好地兼顾了理论与实践，使本书不仅适合数据库开发和管理人员阅读参考，也可以用于高校数据库相关课程的教学。在学习完本书后，你不仅将对MySQL有更加深入的理解，也会对数据库理论有全新的认识，成为一个数据库方面的行家里手。



Charles A. Bell MySQL核心开发人员，目前是Sun公司高级软件工程师；同时也是弗吉尼亚联邦大学的客座教授，主要是为研究生讲授计算机科学课程。他主要从事新兴技术的研究，研究方向包括数据库系统、版本系统、语义网和敏捷软件开发等。

图灵数据库图书阅读路线图



Apress®

本书相关信息请访问：图灵网站 <http://www.turingbook.com>

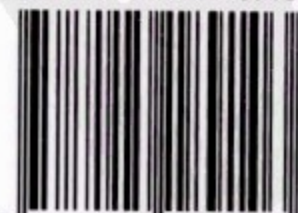
读者/作者热线：(010)51095186

反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/数据库/MySQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-18910-3



9 787115 189103 >

ISBN 978-7-115-18910-3/TP

定价：65.00元